DLD Size-on-MDS Recovery.

Vitaly Fertman

2007-07-13

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 recovery of Size-on-MDS attributes on MDS after a node failure in a cluster, it allows the MDS to re-enable Size-on-MDS caching for inodes with the proper attributes.

1.1 Recovery from a client failure.

The recovery from a client failure is that the MDS performs the client eviction. All the files opened by this client must be closed. If this client was the last writer for a file, MDS must update attributes. As there is no way to get an update from the client, the MDS has to ask the OSTs for the replay logs by itself, and to apply any obtained attribute update to the MDS inodes. After that, the MDS sends out the cancellation requests to the OSTs.

This recovery is transparent for other clients in the cluster, the MDS indicates them it does not have attribute caching enabled on such inodes and the clients obtain them from OSTs.

1.2 Recovery from an MDS failure.

At the time of an MDS failover, the MDS reconstructs the state of inodes during the recovery with clients, they replay to the MDS open, close, WRITE_DONE, SETATTR, etc requests – all that is needed for the inode state reconstruction. The clients are able to proceed their work with the MDS after that.

There also could be missed attribute updates kept on OSTs, in the case there was a client eviction right before the failure and the MDS failed had not finished inode attribute synchronization with OSTs. Thus the MDS synchronizes inodes attributes with OST llogs and the MDS disables caching for the inodes which have an OST with not synchronized llogs in their stripes.

The llog synchronization with OST includes obtaining the pending attribute update llog records from OSTs, getting inodes for each record, obtaining the up-to-date attributes for all these inodes that are not opened for write (during the recovery with clients or by client after that), update the attributes and cancel the processed attribute update logs on the OST's.

1.3 Recovery from the MDS and an OST failure.

At the MDS failover time, the MDS does not know which files need an attribute update. If the MDS cannot obtain the llogs from a failed OST, it does not enable attribute caching for the files that have this OST in their stripes.

1.4 The network failure between a client and the MDS.

The MDS performs the client eviction, however if OSTs have not evicted the client, and the client still has a dirty cache, the client may flush it after the MDS recovers attributes for all the files opened by this client. To sort this out, the MDS recover such files holding the locks over the file objects on the OSTs.

2 Requirements.

After the recovery either the MDS has properly cached SOM attributes for inodes or inodes have indicators their SOM attributes are not valid.

2.1 Architectural requirement.

An IO epoch mechanism is introduced that allows for two attribute retrieval mechanisms:

- OST based when files are active for IO
- MDS based when files are not active for IO

Broadly speaking an IO epoch for a file starts when a file is opened for writing and closes when (i) the file is closed (ii) the IO caches have been flushed. Attribute change may only happen when file is active for IO, i.e. within some IO epoch.

2.2 Functional requirement.

- MDS disables SOM attributes caching for inodes which have an OST with not yet synchronized llogs in their stripes.
- MDS disables SOM attributes caching on inodes whose SOM attributes are to be recovered.
- MDS disables SOM attributes caching on inodes which are not marked as SOM_ENABLED permanently on disk.
- OST performs LLOG_ORIGIN_CONNECT to MDS, MDS reads SOM llogs stored on OSTs, MDS finds inodes which need SOM attribute update by them, MDS puts these inodes into the recovery list, MDS marks this OST as llog-active when all the llogs are obtained from it.
- MDS also puts the inode into the recovery list if a client eviction occurs and this client is the only epoch holder for this inode or this client performs the SOM attribute update for it.
- MDS recovers SOM attributes every inode in the recovery list, updates attributes on the inode and send llog cancels to the proper OSTs.
- MDS remove an inode from the recovery list if a client opens another epoch on it.
- MDS marks files as SOM-ENABLED permanently on disk which it does a SOM attribute update for.

3 Definitions.

IO epoch. The inode on the MDS is in the IO epoch if somewhere in the cluster an IO can be initiated through interaction with OST's.

IO epoch number. The number that uniquely identifies the epoch an inode is in.

4 Use Cases.

1. Client failure.
   - MDS evicts the client;
   - MDT closes all the files opened by this client and closes all the IO epochs as well. If a SOM Attribute Update is not needed (i.e. file was opened for read) or there is another IO epoch holder, the following steps are skipped.

- MDT schedules the object for the recovery;
- MDT wakes up the recovery threads.

2. MDS failover.

   (a) Client replay.

   - Clients replay, resend and send new OPEN, CLOSE, DONE_WRITING, SETATTR requests to MDS;
   - MDT handles them reconstructing the state of the objects it had before the failure;

   (a) OST synchronization.

   - OST llog connects to MDS llog as the llog originator;
   - MDS llog reads the SOM llogs of connected OST;
   - MDS passes the read llogs up through the stack LLOG->OSC->LOV->MDD->MDT.
   - MDT finds an inode for each llog record and schedules the inode for the recovery; if there is no such an object or it is already unlinked, the llog cancel is sent to OST;
   - MDS marks the OST as llog-active once all the OST llogs are obtained;
   - MDS wakes up the recovery threads when an OST llog is completely handled;
   - MDS's recovery thread start working once the client replay is completed, i.e. the inode attribute synchronization is postponed until then.

3. MDS failover at the client failure time.

   - Similar to the MDS failover, but (a) do not believe to SOM attibute updates on CLOSE and DONE_WRITING, i.e. perform the recovery for those inodes anyway, (b) return to alive clients exactly those return codes on their replayed requests which were returned originally (see mdt_epoch_close() logic for details).

4. OST failure.

   - Clients are waiting until OST is up again to flush their caches or to obtain attributes, etc;
   - MDS believes the attributes the clients provides, no check if all the OSTs are alive;
   - MDS schedules the object being closed for the recovery if the client sends the error on the SOM attribute update;

5. OST failover.
    - The same as on OST synchronization on MDS failover.
6. Network problem between client and MDS.
    - The same as the client failure.
7. Recovery of inode SOM attributes.
    - MDS performs the following actions for each inode that is scheduled for the recovery:
    - MDT initiates GETATTR request to OSTs from the object stripe and obtains their attributes, along with the llog cookies;
    - OST performs the getattr under the server-side lock to force all the connected clients to flush their caches in the case of network problems between client and MDS;
    - MDT applies the attributes to the inode and enables SOM on it.
    - MDT writes to the persistent store the obtained attributes;
    - MDD sends a LLOG_CANCEL request to each OST that provided a llog cookie once attributes reaches the disk on MDS;
    - OST removes the LLOG record;
8. Recovery of inode SOM attributes with not yet connected/llog-active OST.
    - MDT detects that not all the OSTs of the object stripe are connected/llog-active;
    - MDT puts the object into the special failure list, which is re-processed on each OST connect;
9. Recovery of inode SOM attributes with permanently disabled OST.
    - MDT detects that an OST of the object stripe is permanently excluded;
    - MDT drops the SOM_ENABLED flag from inode attributes on disk.
10. Recovery failure of inode SOM attributes.
    - MDT initiates GETATTR request to OSTs from the object stripe and is trying to obtain their attributes, along with the llog cookies – this fails due to another OST failure or network problems or similar;
    - MDT drops the SOM_ENABLE flag from inode attributes on disk.
11. Open for write or truncate on an inode which SOM attributes are to be recovered.

- MDT opens a new IO epoch on the inode and removes the object from the recovery list, recovery failure list, etc;
- MDT set back the SOM_ENABLE flag once the SOM attribute update is provided by a client;

12. Open for write or truncate on an inode with not yet llog-active OST.

    - The same as the regular open for write or truncate case.

13. Open for write or truncate on an inode with permanently disabled or failed OST.

    - The same as the regular open for write or truncate case.

5 Functional Specification.

5.1 Data definitions.

1. Inode is marked SOM_ENABLED permanently on disk.

   This flag is mainly for compatibility purpose, what is out of scope of this document. However, recovery also needs the ability to disable SOM caching on an inode for a long period of time: the MDS keeps all the to be recovered inodes in the recovery list; if an OST is unavailable, the MDS just waits when all the OST inode stripes are up and only then it starts recovery. However, we may make an OST permanently unavailable and make it available back later. To not waiting for this OST to get up keeping all the inodes in the memory, we drop the SOM_ENABLED flag for these inodes on disk.

```
#define LDISKFS_SOM_FL              0x00100000
#define LDISKFS_FL_USER_MODIFIABLE 0x001380FF
#define LUSTRE_SOM_FL               LDISKFS_SOM_FL
```

2. Validity flags for moo_attr_get() to indicate the stripe state is needed and an obd getattr is needed are added.

```
enum ma_valid {
    ...
    MA_STRIPE  = (1 << 6),
    MA_OBDATTR = (1 << 7),
};
```

3. Getattr under lock flag.

   A flag to return on clients from MDS which indicates the GETATTR is to be done under lock.

```
#define OBD_MD_FLGETATTRLOCK (0x0000200000000000ULL)
```

4. The SOM recovery flags.

   A flag that identify the inode is in the recovery state is added.

   A flag that is passed on the replay requests that tells the MDS there will be a SOM Attribute Update (AU) is added.

   A flag that is an equivalent of OBD_MD_FLGETATTRLOCK, it tells GETATTR that it is to be done under lock.

```
enum md_op_flags {
    ...
    MF_SOM_RECOV    = (1 << 7),
    MF_SOM_AU       = (1 << 8),
    MF_GETATTR_LOCK = (1 << 9),
};
```

5. the md_attr structure.

   An IO epoch identifier of the current attribute set is added. It is valid when MA_OBDATTR flag is set in ma_valid field.

```
struct md_attr {
    ...
    __u64 ma_ioepoch;
};
```

6. An upcall event to pass new llog record obtained from an OST is added.

```
enum md_upcall_event {
    ...
    MD_SOM_LLOG     = (1 << 3),
};
```

7. New notify events are added, a new LLOG is obtained and LLOG handling is completed:

```
enum obd_notify_event {
    ...
    OBD_NOTIFY_SOM_LLOG,
    OBD_NOTIFY_LLOG_ACTIVE,
};
```

8. The obd_llog_som structure.

   A new structure to pass the llog data up from the llog to the mdt through the whole obd and md stacks.

```
struct obd_llog_som {
    /* If no object is found by the given @ls_fid, the llog record is
     * to be removed. Do it through the given llog context.
     * XXX: it is probably better not to jump over all the stack layers
     * right into the proper llog context, but to call llog_cancel()
     * and going down the stack pick up the right OSC. Should we pass
     * OSC uuid here, add some cancel mdo method and change
     * llog_cancel() interface, have added @uuid there? */
    struct llog_ctxt       *ls_ctxt;
    /* The cookie of the llog record to be removed. */
    struct llog_cookie      ls_cookie;
    /* The fid of the object the llog record belongs to. */
    struct lu_fid           ls_fid;
    /* The IO epoch the llog record belongs to. */
    __u64                   ls_ioepoch;
};
```

9. The OBDO structure.

   A field for mds fid is added. This is a wire format structure, so this change takes place only with OBD_CONNECT_SOM connect flag present. To simplify the compatibility, OBD_INLINESZ is decreased on the size of the mds fid:

```
#define OBD_INLINESZ    64
struct obdo {
    ...
    char           o_inline[OBD_INLINESZ];
    struct lu_fid  o_mds_fid;
};
```

10. Server-side lock.

    OBD_FL_TRUNCLOCK is renamed to OBD_FL_SRVLOCK to not be limited with the truncate only. Used for getattr under the server-side lock from MDS. It requires OBD_CONNECT_TRUNCLOCK connection flag on truncate and OBD_CONNECT_SOM on getattr.

11. The llog_size_change_rec structure.

    It keeps the lu_fid structure now instead of ll_fid.

5.2 MD Object Interface.

1. int (*mu_upcall)(const struct lu_env *env, struct md_device *md, enum md_upcall_event ev, void *data);

   New parameter @data is added to pass the llog record information.

5.3 OBD Interface.

1. int (*o_llog_connect)(struct obd_export *exp, struct llogd_conn_body *conn, struct obd_uuid *uuid);

   Another parameter @uuid is added to let LOV to handle LLOG_ORIGIN_CONNECT for the proper OSC.

2. int (*onu_upcall)(struct obd_device *host, struct obd_device *watched, enum obd_notify_event ev, void *owner, void *data);

   A new parameter @data is added to pass the obtained llog record.

3. int (*o_stripe_state)(struct obd_export *, struct lov_stripe_md *);

   Get the state of the stripe, in particular if the OSTs are connected and activated, if their llogs are activated.

6 Logical Specification.

6.1 MDT.

6.1.1 MDT Data Definitions.

1. mdt_device.

   @mdt_ioepoch is renamed to @mdt_ioepoch_seq, this is the IO epoch sequenser.

   A list of SOM recovery threads with wait-queue and with a spinlock for the synchronization are added.

   A list of objects that needs their SOM attributes to be recovered and another list for objects whose recovery failed are also added. Another list of fid's which objects are to be recovered is added. They all are guarded with mdt_ioepoch_lock.

   A flag that another OST is llog-active is added, this is an indicator for SOM recovery threads to process the failed recovery list again.

```
struct mdt_device {
    ...
    struct list_head mdt_som_threads;
    spinlock_t mdt_som_lock; /* XXX: use mdt_ioepoch_lock instead? */
    cfs_waitq_t mdt_som_waitq;
    struct list_head mdt_som_list;
    struct list_head mdt_som_failed;
    struct list_head mdt_fid_list;
};
```

2. mdt_object.

   The mdt recovery list is added into the object.

```
struct mdt_object {
    ...
    struct list_head mot_recovery;
};
```

3. mdt_thread_info

   @mti_replayepoch is renamed to @mti_recov_epoch. Besides keeping the replayed ioepoch it is also used for keeping the ioepoch the object recovery is performed for.

   An md_attr structure is added into mti_u for keeping the temporary attributes.

```
struct mdt_thread_info {
    ...
    union {
        ...
        struct md_attr attr;
    } mti_u;
};
```

4. mdt_som_fid

   The special structure to keep the list of fids, to be precise obd_llog_som structures, which objects to be recovered.

```
struct mdt_som_fid {
    struct list_head        sf_list;
    struct obd_llog_som     sf_llog;
};
```

6.1.2 SOM Caching basis.

1. int mdt_object_is_som_enabled(struct mdt_object *mo)

   Besides the existent checks that objects is not in IO Epoch and there is no pending SOM attribute update, also checks that the object is not in the recovery state.

```
return (mo->mot_ioepoch == 0) && !(mo->mot_flags & MF_SOM_RECOV);
```

2. void mdt_pack_size2body(struct mdt_thread_info *info, struct mdt_object *o)

Checks that SOM caching is enabled on the object. Besides the existent checks the object is regular file, there is no opened epoch on it nor waiting for SOM attribute update, the following checks are added:

- MA_STRIPE attributes are valid, i.e. all the OSTs of the stipe are llog-active.
- the object has LUSTRE_SOM_FL flag in its attributes on disk, i.e. attibutes were updated;
- the object is not in the recovery state.

```
LASSERT(ma->ma_attr.la_valid & LA_MODE);
b = req_capsule_server_get(&info->mti_pill, &RMF_MDT_BODY);
if (!(mdt_conn_flags(info) & OBD_CONNECT_SOM) ||
    !S_ISREG(la->la_mode) || !(ma->ma_valid & MA_STRIPE) ||
    !mdt_object_is_som_enabled(o) || !(la->la_valid & LA_FLAGS) ||
    !(la->la_flags & LUSTRE_SOM_FL))
        return;
b->valid |= OBD_MD_FLSIZE | OBD_MD_FLBLOCKS;
b->size = la->la_size;
b->blocks = la->la_blocks;
```

3. static int mdt_getattr_name_lock(struct mdt_thread_info *info, struct mdt_lock_handle *lhc, __u64 child_bits, struct ldlm_reply *ldlm_rep)

   To have MA_STRIPE attributes valid in mdt_pack_size2body() we need to getattr them beforehand.

```
ma_need = 0;
lock = ldlm_handle2lock(&lhc->mlh_reg_lh);
if (lock) {
    /* Get MA_STRIPE attributes if update lock is given. */
    if (lock->l_policy_data.l_inodebits.bits & MDS_INODELOCK_UPDATE)
        ma_need = MA_STRIPE;
}
mdt_set_capainfo(info, 1, child_fid, BYPASS_CAPA);
rc = mdt_getattr_internal(info, child, ma_need);
if (unlikely(rc != 0)) {
    mdt_object_unlock(info, child, lhc, 1);
} else {
    /* Debugging code. */
    res_id = &lock->l_resource->lr_name;
    LDLM_DEBUG(lock, "Returning lock to client\n");
    LASSERTF(fid_res_name_eq(mdt_object_fid(child),
             &lock->l_resource->lr_name),
             "Lock res_id: %lu/%lu/%lu, Fid: "DFID".\n",
             (unsigned long)res_id->name[0],
             (unsigned long)res_id->name[1],
```

```
                (unsigned long)res_id->name[2],
                PFID(mdt_object_fid(child)));
        mdt_pack_size2body(info, child);
    }
    if (lock)
        LDLM_LOCK_PUT(lock);
```

4. static int mdt_getattr_internal(struct mdt_thread_info *info, struct mdt_object *o, __u64 ma_need)

   It gets another parameter @ma_valid, which tells if more attributes are to be obtained, MA_STRIPE is passed from mdt_getattr_name_lock() here.

   ```
   ma->ma_need |= ma_need;
   ```

5. int mdt_epoch_open(struct mdt_thread_info *info, struct mdt_object *o)

   When opens a new epoch, take the replayed epoch or the new one. The replayed epoch may be less than already set in the object, if OST has sent its SOM llogs. Do not set ioepoch into the object if so.

   For a new epoch, drop the MD_SOM_RECOV flag and set MD_SOM_CHANGE one to force the client to obtain the attributes on the epoch close.

   For the recovery case, always mark the object as to be recovered. If some client will update SOM attributes for us, we will drop it, otherwise the recovery will be performed.

   Remove the object from the recovery queue, while an IO epoch is opened, no recovery takes place.

   ```
   if (mdt_epoch_opened(o)) {
       ...
   } else {
       ...
       if (mdt->mdt_ioepoch_seq < info->mti_recov_epoch)
           mdt->mdt_ioepoch_seq = info->mti_recov_epoch;
       else if (info->mti_recov_epoch == 0)
           mdt->mdt_ioepoch_seq++;
       if (!info->mti_recov_epoch) {
           LASSERT(o->mot_ioepoch < mdt->mdt_ioepoch_seq);
           epoch = o->mot_ioepoch = mdt->mdt_ioepoch_seq;
           if (o->mot_flags & MF_SOM_RECOV) {
               o->mot_flags &= ~MF_SOM_RECOV;
               o->mot_flags |= MF_SOM_CHANGE;
           }
       } else {
           if (o->mot_ioepoch < info->mti_recov_epoch)
   ```

```
                epoch = o->mot_ioepoch = info->mti_recov_epoch;
            else
                epoch = info->mti_recov_epoch;
            o->mot_flags |= MF_SOM_RECOV;
        }
        list_del_init(&o->mot_recovery);
        ...
    }
```

6. static int mdt_epoch_close(struct mdt_thread_info *info, struct mdt_object *o, int llog_active)

   A new parameter @llog_active is added, it identifies if the stripe is llog-active.

   If an object is marked as MF_SOM_RECOV or this is a replay request, we cannot believe into its attributes, because we cannot guarantee there was no other client before the failure. Otherwise it could happen another client succeeded to open/write/close the file after this client did it and disappeared while mds is getting recovered – MDS does not remember the object state properly, precisely that there was another IO epoch holder, and therefore cannot believe in CLOSE and DONE_WRITE attributes.

   If this is not a replay request and object is not marked as MF_SOM_RECOV, there is a danger that not all the OST are llog-acvite and therefore we cannot believe into attributes as well, the client is asked for attributes under lock (see mdt_mfd_close() logic as well).

   In addition to the previous code, the following is added:

   If an eviction occurs, and there is no more IO epoch holders, schedule the object for the recovery;

   If epoch closes and this is a replay case, get an info from the replayed request if is supposed to send an Attibute Update and reconstruct @rc = -EAGAIN if so.

   If epoch closes and this is not replay and the request IO epoch matches the object's one, re-ask client for AU in the following cases: the previous IO Epoch holder has changed the attibutes, the object is already marked as MF_SOM_RECOV, the current client changed attributes but has not provided an AU, not all the stripe is llog-active.

   If we ask the client for an AU for the uptodate IO epoch, we do not need MF_SOM_RECOV flag on the object anymore, drop it.

   If we do not ask the client for an AU or this is a replay case for a previous IO epoch, however the object is marked as MF_SOM_RECOV, shedule the object for the recovery.

   Mark the object as MF_SOM_CHANGE not only if the current request brings this flag, but if we ask the client for an AU, i.e. in the replay case

we may lose another IO epoch holder. Mark the object before the recovery as well.

```
if (!(mdt_conn_flags(info) & OBD_CONNECT_SOM) ||
    !S_ISREG(lu_object_attr(&o->mot_obj.mo_lu)))
    RETURN(0);
/* Epoch closes only if client tells about it or eviction
 * occures. */
eviction = (req == NULL ? 1 : 0);
if (!eviction && !(info->mti_epoch->flags & MF_EPOCH_CLOSE))
    RETURN(1);
LASSERT(o->mot_epochcount);
spin_lock(&info->mti_mdt->mdt_ioepoch_lock);
o->mot_epochcount--;
opened = mdt_epoch_opened(o);
CDEBUG(D_INODE, "Closing epoch "LPU64" on "DFID". Count %d\n",
       o->mot_ioepoch, PFID(mdt_object_fid(o)),
       o->mot_epochcount);
if (eviction) {
    o->mot_flags |= MF_SOM_CHANGE;
    /* If eviction occured and no other epoch holders, set
     * MF_SOM_RECOV flag and return EAGAIN to show the caller
     * the recovery is needed. */
    if (!opened)
        mdt_som_queue_recovery(info, o, 0);
    spin_unlock(&info->mti_mdt->mdt_ioepoch_lock);
    RETURN(0);
}
ioepoch = info->mti_epoch->ioepoch;
LASSERT(ioepoch <= o->mot_ioepoch);
replay = lustre_msg_get_flags(req->rq_reqmsg) & MSG_REPLAY;
achange = (info->mti_epoch->flags & MF_SOM_CHANGE);
if (!opened) {
    /* Epoch ends. Is an Size-on-MDS update needed? */
    LASSERT(list_empty(&o->mot_recovery));
    if (replay) {
        /* Get an info from the replayed request if client
         * is supposed to send an Attibute Update. IF so,
         * the previous @rc was -EAGAIN, reconstruct it. */
        if (info->mti_epoch->flags & MF_SOM_AU)
            rc = -EAGAIN;
    } else if (ioepoch == o->mot_ioepoch) {
        /* The epoch is closed for the current IO epoch. */
        if (o->mot_flags & MF_SOM_RECOV)
            /* If the object is marked for recovery,
             * re-ask the client for attributes. It is
```

```
                 * needed for the cases when it was opened
                 * on replay and another epoch holder
                 * disappeared -- we cannot believe these
                 * attributes. */
                rc = -EAGAIN;
            else if (o->mot_flags & MF_SOM_CHANGE)                                  /
                 * Do not believe to the current Size-on-MDS
                 * update, re-ask client. */
                rc = -EAGAIN;
            else if (!(la->la_valid & LA_SIZE) && achange)
                /* Attributes were changed by the last writer
                 * only but no Size-on-MDS update is received.*/
                rc = -EAGAIN;
            else if (!llog_active)
                /* Not all the OST are llog-active yet,
                 * ask the client for the attributes under
                 * the lock (see mdt_mfd_close() as well). */
                rc = -EAGAIN;
        }
        if (o->mot_flags & MF_SOM_RECOV) {
            o->mot_flags |= MF_SOM_CHANGE;
            if (rc == -EAGAIN && ioepoch == o->mot_ioepoch)
                o->mot_flags &= ~MF_SOM_RECOV;
            else
                mdt_som_queue_recovery(info, o, 0);
        } else {
            /* For the object not marked for recovery,
             * the epoch must match. */
            LASSERT(ioepoch == o->mot_ioepoch);
        }
    }
    if (achange || rc == -EAGAIN)
        o->mot_flags |= MF_SOM_CHANGE;         spin_unlock(&info->mti_mdt->mdt_ioepoch_
    if ((rc == 0) && !opened && achange && !replay) {
        /* Epoch ends and relable SOM attributes are obtained,
         * update them. */
        rc = mdt_som_update(info, o);
        /* Avoid the following setattrs of these attributes, e.g.
         * for atime update, as attributes are already written. */
        info->mti_attr.ma_valid = 0;
    } else if (rc == -EAGAIN) {
        /* Avoid the following setattrs as well if this client will
         * perform the attribute update for us. However, if rc == 0
         * and there is another IO epoch holder, write atime,
         * otherwise this client atime change be lost. */
        info->mti_attr.ma_valid = 0;
```

```
}
RETURN(rc);
```

7. static int mdt_som_close(struct mdt_thread_info *info, struct mdt_object *o)

   Checks if an eviction occurred for a regular file or this is a SOM attribute update for no size (i.e. the client failed to obtain the attributes) and there is no more ioepoch holders left, schedule the object for the recovery if so.

```
eviction = (mdt_info_req(info) == NULL ? 1 : 0);
if (!S_ISREG(lu_object_attr(&o->mot_obj.mo_lu)))
    RETURN(0);
if (info->mti_epoch && (info->mti_epoch->flags & MF_SOM_CHANGE) &&
    !(info->mti_attr.ma_attr.la_valid & LA_SIZE))
        nosize = 1;
if (!eviction && !nosize)
    RETURN(0);
spin_lock(&info->mti_mdt->mdt_ioepoch_lock);
opened = mdt_epoch_opened(o);
if (!opened)
    mdt_som_queue_recovery(info, o, 0);
spin_unlock(&info->mti_mdt->mdt_ioepoch_lock);
RETURN(0);
```

8. int mdt_mfd_close(struct mdt_thread_info *info, struct mdt_file_data *mfd)

   Figure out if the object's stripe is completely llog-active, i.e. get MA_STRIPE attributes, and pass this info into mdt_epoch_close().

   Checks if the recovery is needed for an objects that has been waiting for SOM attribute update, call mdt_som_close() and EAGAIN will be the right indicator of this.

   If the client is asked for an attribute update but the object stripe is not llog-active, ask client to gather object SOM attributes under the lock – pass OBD_MD_FLGETATTRLOCK in reply.

```
mti_ma = &info->mti_u.attr;
mti_ma->ma_need = MA_INODE | MA_LOV | MA_STRIPE | ma->ma_need;
rc = mo_attr_get(info->mti_env, next, mti_ma);
if (rc)
    RETURN(rc);
eclose = mode & (FMODE_EPOCH | FMODE_WRITE | FMODE_EPOCHLCK);
if ((mode & FMODE_WRITE) || (mode & FMODE_EPOCHLCK))
    mdt_write_put(info->mti_mdt, o);
else if (mode & MDS_FMODE_EXEC)
    mdt_write_allow(info->mti_mdt, o);
```

```
else if ((mode & FMODE_SOM))
    ret = mdt_som_close(info, o);
if (eclose)
    ret = mdt_epoch_close(info, o, mti_ma->ma_valid & MA_STRIPE);
...
if (!MFD_CLOSED(mode))
    rc = mo_close(info->mti_env, next, ma);
if ((ret == -EAGAIN) && (mti_ma->ma_valid & MA_INODE) &&
    (mti_ma->ma_attr.la_nlink == 0))
    ret = 0;
if (ret == -EAGAIN)) {
    /* If MA_STRIPE is not present, not all the OST are
     * synced, tell the client to perform GETATTR under the lock. */
    if (!(mti_ma->ma_valid & MA_STRIPE)) {
        struct mdt_body *repbody =
            req_capsule_server_get(&info->mti_pill, &RMF_MDT_BODY);
            repbody->valid |= OBD_MD_FLGETATTRLOCK;
    }
}
```

9. void mdt_object_som_enable(struct mdt_thread_info *info, struct mdt_object *mo, __u64 ioepoch)

   New parameter @ioepoch is added. Enables SOM caching on the object if its ioepoch matches to @ioepoch instead of the @info->mti_epoch->ioepoch. This parameter is still usually @info->mti_epoch->ioepoch except for the new recovery cases covered in this DLD.

   To the addition to previous actions, If epoch matches the object is removed from the recovery list. All the flags are just zeroed as before.

   ```
   if (ioepoch == mo->mot_ioepoch) {
       ...
       mo->mot_flags = 0;
       list_del_init(&mo->mot_recovery);
   }
   ```

10. int mdt_attr_set(struct mdt_thread_info *info, struct mdt_object *mo, int flags)

    The recovery cases handling is added – sets obtained from the OSTs attributes for @info->mti_recov_epoch epoch.

    ```
    if (mdt_info_req(info) == NULL) {
        ioepoch = info->mti_recov_epoch;
        ASSERT(ioepoch > 0);
        som_update = 1;
    } else if (info->mti_epoch) {
        ioepoch = info->mti_epoch->ioepoch;
    ```

```
    som_update = (info->mti_epoch->flags & MF_SOM_CHANGE);
}
/* Use @ioepoch instead of @info->mti_epoch->ioepoch further down. */
if (som_update && (ioepoch != mo->mot_ioepoch))
    RETURN(0);
...
if (som_update && (ioepoch != mo->mot_ioepoch))
    GOTO(out_unlock, rc = 0);
...
if (som_update) {
    ...
    mdt_object_som_enable(info, mo, ioepoch);
}
```

11. static int mdt_reint_setattr(struct mdt_thread_info *info, struct mdt_lock_handle *lhc)

    We do believe into attributes when the object is marked as MF_RECOV or if this is a replay RPC, because even if some client succeeded to open/write/close the file after the epoch closed and disappeared while mds is getting recovered, i.e. MDS does not remember this IO epoch holder, this is a new IO epoch holder, whereas these attributes are tagged with the previous IO epoch number. Thus no special logic is added here.

    If a SOM attribute update comes with no size change, this is an indicator about an error occurred on the client during obd_getattr(). Put this object to the recovery list if so.

```
if (info->mti_epoch && (info->mti_epoch->flags & MF_SOM_CHANGE))
    som_change = 1;
if (!som_change || (ma->ma_attr.la_valid & LA_SIZE)) {
    rc = mdt_attr_set(info, mo, rr->rr_flags);
    if (rc)
        GOTO(out_put, rc);
}
if (som_change) {
    ...
    mdt_mfd_close(info, mfd);
    if (!(ma->ma_attr.la_valid & LA_SIZE))
        cfs_waitq_signal(&mdt->mdt_som_waitq);
}
...
mdt_empty_transno(info);
out:
return rc;
```

6.1.3 SOM recovery.

1. Data to be passed to a particular som recovery thread.

```
struct mdt_som_th_data {
    struct ptlrpc_thread *thread;
    struct mdt_device *mdt;
    char *name;
};
```

2. static int mdt_som_thread_main(void *arg)

   The SOM recovery handler.

   - Every thread starts doing its work only when the client recovery is completed;
   - Every thread is waiting for an event on @mdt->mdt_som_waitq, and checks if @mdt->mdt_som_list or @mdt->mdt_som_failed or @mdt->mdt_fid_list list is not empty or the thread state has changed to SVC_STOPPING.
   - A thread starts mdt_som_fids_recover() for every fid in @mdt->mdt_fid_list list and starts mdt_som_objs_recover() for every object in the @mdt->mdt_som_list .
   - If @mdt->mdt_som_list is empty but @mdt->mdt_som_failed is not, and an OST is llog-active, move the @mdt->mdt_som_failed to @mdt->mdt_som_list and try to recover these objects again.

```
msd    = (struct mdt_som_th_data *)arg;
mdt    = msd->mdt;
thread = msd->thread;
cfs_daemonize(msd->name);
thread->t_flags = SVC_RUNNING;
CDEBUG(D_INFO, "SOM recovery thread starting, process %d\n",
       cfs_curproc_pid());
rc = lu_env_init(&env, NULL, LCT_MD_THREAD);
if (rc)
    RETURN(rc);
thread->t_env = &env;
env.le_ctx.lc_thread = thread;
info = lu_context_key_get(&env.le_ctx, &mdt_thread_key);
LASSERT(info != NULL);
memset(info, 0, sizeof *info);
info->mti_env = &env;
info->mti_mdt = mdt;
ma = &info->mti_attr;
lmm_size = ma->ma_lmm_size = mdt->mdt_max_mdsize;
```

```
cookie_size = ma->ma_cookie_size = mdt->mdt_max_cookiesize;
OBD_ALLOC(ma->ma_lmm, lmm_size);
OBD_ALLOC(ma->ma_cookie, cookie_size);
if (ma->ma_lmm == NULL || ma->ma_cookie == NULL)
    GOTO(out, rc = -ENOMEM);
cfs_waitq_signal(&mdt->mdt_som_waitq);
l_wait_event(info->mti_exp->exp_obd->obd_next_transno_waitq,
             is_recovery_completed(info->mti_exp->exp_obd), &lwi);
while (1) {
    static atomic_t handlers = { 0 };
    static atomic_t connected = { 0 };
    l_wait_event(mdt->mdt_som_waitq,
                 thread->t_flags & SVC_STOPPING ||
                 !list_empty(&mdt->mdt_som_list) ||
                 !list_empty(&mdt->mdt_som_failed) ||
                 !list_empty(&mdt->mdt_fid_list), &lwi);
    atomic_inc(&handlers);
    if (mdt_som_fids_recover(info, thread))
        atomic_set(&connected, 1);
    mdt_som_objs_recover(info, thread);
    atomic_dec(&handlers);
    spin_lock(&mdt->mdt_ioepoch_lock);
    if (atomic_read(&handlers) == 0 &&
        info->mti_mdt->mdt_som_connected &&
        !list_empty(&mdt->mdt_som_failed) &&
        ((atomic_read(&connected) || thread->t_flags & SVC_STOPPING))
    {
        atomic_set(&connected, 0);
        list_splice_init(&mdt->mdt_som_failed, &mdt->mdt_som_list);
        spin_unlock(&mdt->mdt_ioepoch_lock);
        cfs_waitq_signal(&info->mti_mdt->mdt_som_waitq);
        continue;
    }
    if (!list_empty(&mdt->mdt_fid_list) &&
        thread->t_flags & SVC_STOPPING) {
        spin_unlock(&mdt->mdt_ioepoch_lock);
        continue;
    }
    spin_unlock(&mdt->mdt_ioepoch_lock);
    if (thread->t_flags & SVC_STOPPING)
        break;
}
thread->t_flags = SVC_STOPPED;
cfs_waitq_signal(&mdt->mdt_som_waitq);
CDEBUG(D_INFO, "SOM recovery thread exiting, process %d\n",
       cfs_curproc_pid());
```

```
EXIT;
out:
info->mti_mdt = NULL;
if (lmm_size) {
    OBD_FREE(ma->ma_lmm, lmm_size);
    ma->ma_lmm = NULL;
}
if (cookie_size) {
    OBD_FREE(ma->ma_cookie, cookie_size);
    ma->ma_cookie = NULL;
}
lu_env_fini(&env);
return rc;
```

3. static int is_recovery_completed(struct obd_device *obd)

   Check that the recovery with clients is completed.

```
spin_lock_bh(&obd->obd_processing_task_lock);
rc = obd->obd_recovering == 0;
spin_unlock_bh(&obd->obd_processing_task_lock);
return rc;
```

4. static int mdt_som_fids_recover(struct mdt_thread_info *info, + struct ptlrpc_thread *thread)

   Walk through the list of fids which objects are to be recovered, find an object for each and schedule the object for the recovery. If no object can be found, cancel the llog record the fid was obtained from.

```
spin_lock(&mdt->mdt_ioepoch_lock);
while (!list_empty(&mdt->mdt_som_list)) {
    sfid = list_entry(mdt->mdt_fid_list.next,
                      struct mdt_som_fid, sf_list);
    list_del_init(&sfid->sf_list);
    if (thread->t_flags & SVC_STOPPING) {
        OBD_FREE_PTR(sfid);
        continue;
    }
    spin_unlock(&mdt->mdt_ioepoch_lock);
    obj = mdt_object_find(info->mti_env, info->mti_mdt,
                          &sfid->sf_llog.ls_fid);
    if (IS_ERR(obj))
        obj = NULL;
    spin_lock(&info->mti_mdt->mdt_ioepoch_lock);
    if (obj && mdt_object_exists(obj)) {
        mdt_som_queue_recovery(info, obj, sfid->sf_llog.ls_ioepoch);
        count++;
```

```
        } else {
            /* XXX: not correct to call llog methods here. To be
             * fixed with the new llog api. */
            llog_cancel(sfid->sf_llog.ls_ctxt, NULL, 1,
                        &sfid->sf_llog.ls_cookie, 0);
        }
        if (obj)
            mdt_object_put(info->mti_env, obj);
        OBD_FREE_PTR(sfid);
    }
    spin_unlock(&info->mti_mdt->mdt_ioepoch_lock);
    RETURN(count);
```

5. static int mdt_som_objs_recover(struct mdt_thread_info *info, + struct ptlrpc_thread *thread)

   Walk through the list of objects to be recovered and perform the recovery for each.

   If the recovery returns EAGAIN, it means not all the OST are llog-active yet, put the object into the mdt_som_failed list, if it returns another error call mdt_som_invalid() to drop SOM_ENABLE flag in the inode attribtes.

```
    spin_lock(&mdt->mdt_ioepoch_lock);
    while (!list_empty(&mdt->mdt_som_list)) {
        obj = list_entry(mdt->mdt_som_list.next,
                         struct mdt_object, mot_recovery);
        list_del_init(&obj->mot_recovery);
        if (thread->t_flags & SVC_STOPPING) {
            mdt_object_put(info->mti_env, obj);
            continue;
        }
        LASSERT(obj->mot_flags & MF_SOM_RECOV);
        info->mti_recov_epoch = obj->mot_ioepoch;
        spin_unlock(&mdt->mdt_ioepoch_lock);
        LASSERT(info->mti_recov_epoch);
        rc = mdt_som_recover(info, obj);
        if (rc && rc != -EAGAIN) {
            if (!(ma->ma_valid & MA_INODE))
                RETURN(rc);
            mdt_som_invalid(info, obj);
        }
        ma->ma_lmm_size = lmm_size;
        ma->ma_cookie_size = cookie_size;
        spin_lock(&mdt->mdt_ioepoch_lock);
        if (rc == -EAGAIN && list_empty(&obj->mot_recovery))
            list_add(&obj->mot_recovery, &mdt->mdt_som_failed);
```

```
        else
            mdt_object_put(info->mti_env, obj);
    }
    spin_unlock(&mdt->mdt_ioepoch_lock);
    RETURN(0);
```

6. static int mdt_som_recover(struct mdt_thread_info *info, struct mdt_object *obj)

   Performs the recovery for @obj, if a regular file, for @info->mti_recov_epoch epoch: call mo_attr_get() for MA_INODE | MA_LOV | MA_STRIPE | MA_OBDATTR attributes, i.e. check if all the OST from the stripe are llog-active and if they are, do OBD_GETATTR on OSTs.

```
if (!info->mti_recov_epoch ||
    !S_ISREG(lu_object_attr(&obj->mot_obj.mo_lu)))
    RETURN(0);
ma->ma_need = MA_INODE | MA_LOV | MA_STRIPE | MA_OBDATTR;
ma->ma_valid = 0;
ma->ma_cookie_som_size = ma->ma_cookie_size;
ma->ma_cookie_som = ma->ma_cookie;
ma->ma_ioepoch = info->mti_recov_epoch;
rc = mo_attr_get(info->mti_env, mdt_object_child(obj), ma);
if (rc)
    RETURN(rc);
if (ma->ma_valid & MA_OBDATTR)
    rc = mdt_attr_set(info, obj, 0);
RETURN(rc);
```

7. static int mdt_som_invalid(struct mdt_thread_info *info, struct mdt_object *obj)

   Drops the LUSTRE_SOM_FL flag in the inode attributes.

```
LASSERT(ma->ma_valid & MA_INODE);
ma->ma_attr.la_valid = LA_FLAGS;
ma->ma_attr.la_flags &= ~LUSTRE_SOM_FL;
/* XXX: should not we take the lock beforehand? */
rc = mdt_attr_set(info, obj, 0);
RETURN(rc);
```

8. void mdt_som_queue_recovery(struct mdt_thread_info *info, struct mdt_object *obj, __u64 ioepoch)

   Marks the object @obj as needed the recovery, and put it to the recovery list. It is called under @mdt_ioepoch_lock lock held. @ioepoch indicates the epoch number the objects needs the recovery for. Do it only if:

   - the object is not in the given or larger io epoch yet; If @ioepoch == 0, we always put the object into the recovery list;

- there is no IO epoch holders;
- the object is not in the recovery list yet;

We always mark the object as MF_SOM_RECOV to perform the recovery later.

```
if (ioepoch && obj->mot_ioepoch >= ioepoch)
    return;
obj->mot_flags |= MF_SOM_RECOV;
if (ioepoch)
    obj->mot_ioepoch = ioepoch;
if (obj->mot_epochcount == 0 && list_empty(&obj->mot_recovery)) {
    list_add(&obj->mot_recovery, &mdt->mdt_som_list);
    mdt_object_get(info->mti_env, obj);
}
```

9. static int mdt_destroy_export(struct obd_export *export)

   When all the objects on the destroyed export are handled, wakes up the recovery threads.

```
while (!list_empty(&med->med_open_head)) {
    ...
}
cfs_waitq_signal(&mdt->mdt_som_waitq);
```

10. static int mdt_som_start_thread(struct mdt_device *mdt, char *name, int id)

    Start a SOM recovery thread.

```
struct mdt_som_th_data msd;
OBD_ALLOC_PTR(msd.thread);
if (msd.thread == NULL)
    RETURN(-ENOMEM);
msd.mdt = mdt;
msd.name = name;
msd.thread->t_id = id;
spin_lock(&mdt->mdt_som_lock);
list_add(&msd.thread->t_link, &mdt->mdt_som_threads);
spin_unlock(&mdt->mdt_som_lock);
cfs_waitq_init(&mdt->mdt_som_waitq);
rc = cfs_kernel_thread(mdt_som_thread_main, &msd,
                       (CLONE_VM | CLONE_FILES | CLONE_FS));
if (rc < 0) {
    CERROR("cannot start thread %s, rc = %d\n", name, rc);
    spin_lock(&mdt->mdt_som_lock);
    list_del(&msd.thread->t_link);
```

```
        spin_unlock(&mdt->mdt_som_lock);
        OBD_FREE(msd.thread, sizeof(*thread));
        RETURN(rc);
    }
    l_wait_event(mdt->mdt_som_waitq,
                 msd.thread->t_flags & SVC_RUNNING, &lwi);
    RETURN(0);
```

11. int mdt_som_start_threads(struct mdt_device *mdt)

    Start the SOM recovery threads.

```
    cfs_waitq_init(&mdt->mdt_som_waitq);
    for (i = 0; i < SOM_THREAD_COUNT; i++) {
        sprintf(name, "som_%02d", i);
        rc = mdt_som_start_thread(mdt, name, i);
        if (rc) {
            mdt_som_stop_threads(mdt);
            break;
        }
    }
    RETURN(rc);
```

12. void mdt_som_stop_threads(struct mdt_device *mdt)

    Stop the SOM recovery threads.

```
    spin_lock(&mdt->mdt_som_lock);
    while (!list_empty(&mdt->mdt_som_threads)) {
        thread = list_entry(mdt->mdt_som_threads.next,
                            struct ptlrpc_thread, t_link);
        list_del(&thread->t_link);
        spin_unlock(&mdt->mdt_som_lock);
        if (thread->t_flags & SVC_RUNNING) {
            thread->t_flags = SVC_STOPPING;
            cfs_waitq_signal(&mdt->mdt_som_waitq);
            l_wait_event(thread->t_ctl_waitq,
                         thread->t_flags & SVC_STOPPED, &lwi);
        }
        OBD_FREE(thread, sizeof(*thread));
        spin_lock(&mdt->mdt_som_lock);
    }
    spin_unlock(&mdt->mdt_som_lock);
```

13. static int mdt_init0(const struct lu_env *env, struct mdt_device *m, struct lu_device_type *ldt, struct lustre_cfg *cfg)

    Starts SOM recovery threads.

```
rc = mdt_som_start_threads(m);
```

14. static void mdt_fini(const struct lu_env *env, struct mdt_device *m)

    Stop SOM recovery threads.

    ```
    mdt_som_stop_threads(m);
    ```

15. static int mdt_upcall(const struct lu_env *env, struct md_device *md, enum md_upcall_event ev, void *data)

    Handle the new MD_SOM_LLOG event:

    ```
    case MD_SOM_LLOG:
        rc = mdt_som_llog_add(m, (struct obd_llog_som *)data);
        break;
    ```

16. static int target_recovery_thread(void *arg)

    Wake up the SOM threads when the client recovery is cmpleted.

    ```
    spin_lock_bh(&obd->obd_processing_task_lock);          obd->obd_recovering = obd->o
    wake_up(&obd->obd_next_transno_waitq);
    ```

6.1.4 LLOG handling.

1. mdt_llog_ops[]

   The LLOG_ORIGIN_CONNECT method is installed.

   ```
   #define OBD_FAIL_LLOG_ORIGIN_CONNECT     0x1300
   #define DEF_LLOG_HNDL(flags, name, fn)                    \
       DEF_HNDL(LLOG, ORIGIN_HANDLE_CREATE, , flags, name,
                fn, &RQF_LLOG_ ## name)
   static struct mdt_handler mdt_llog_ops[] = {
       DEF_LLOG_HNDL(0, ORIGIN_CONNECT, mdt_llog_handle_connect)
   };
   ```

2. static int mdt_llog_handle_connect(struct mdt_thread_info *info)

   The MDT handler for LLOG_ORIGIN_CONNECT rpc. It performs the initialization of replicator llogs, reading originator llogs and processing them on the replicator.

   ```
   req = mdt_info_req(info);
   body = lustre_msg_buf(req->rq_reqmsg, 1, sizeof(*body));
   uuid = &req->rq_import->imp_conn_current->oic_uuid;
   req->rq_status = obd_llog_connect(req->rq_export, body, uuid);
   rc = lustre_pack_reply(req, 1, NULL, NULL);
   RETURN(rc);
   ```

3. static int mdt_som_llog_add(struct mdt_device *mdt, struct obd_llog_som *lsom)

   The llog recovery thread obtains a llog record to be processed and passes @lsom, actually the object fid with some more data, built on it here. Schedule this fid for the recovery. If the fid is NULL, the llog processing is completed, wake up the SOM recovery threads.

```
if (lsom == NULL) {
    cfs_waitq_signal(&mdt->mdt_som_waitq);
    RETURN(0);
}
if (!fid_is_sane(&lsom->ls_fid)) {
    CERROR("Invalid fid: "DFID"\n", PFID(&lsom->ls_fid));
    RETURN(-EINVAL);
}
OBD_ALLOC_PTR(sfid);
if (sfid == NULL)
    RETURN(-ENOMEM);
CFS_INIT_LIST_HEAD(&sfid->sf_list);
sfid->sf_llog = *lsom;
spin_lock(&mdt->mdt_ioepoch_lock);
list_add(&sfid->sf_list, &mdt->mdt_fid_list);
spin_unlock(&mdt->mdt_ioepoch_lock);
RETURN(0);
```

6.2 MDD.

1. static int mdd_stripe_state(const struct lu_env *env, struct md_object *obj, struct md_attr *ma)

   Having MA_LOV in @ma, check the state of the stripe.

```
LASSERT(ma->ma_valid & MA_LOV);
if (IS_ERR(mds->mds_osc_obd))
    RETURN(PTR_ERR(mds->mds_osc_obd));
rc = obd_unpackmd(mds->mds_osc_exp, &lsm, ma->ma_lmm,
                  ma->ma_lmm_size);
if (rc < 0)
    RETURN(rc);
rc = obd_checkmd(mds->mds_osc_exp,
                 mds->mds_osc_obd->obd_self_export, lsm);
if (rc)
    GOTO(out, rc);
rc = obd_stripe_state(mds->mds_osc_exp, &lsm);
if (rc == 0) {
    /* obd stripes are connected and synched */
```

```
    ma->ma_valid |= MA_STRIPE;
}
EXIT;
out:
obd_free_memmd(mds->mds_osc_exp, &lsm);
return rc;
```

2. static int mdd_attr_get(const struct lu_env *env, struct md_object *obj, struct md_attr *ma)

   Perform the getattr for 2 more cases: MA_STRIPE and MA_OBDATTR, i.e. check the state of the object stripe and perform the obd_getattr for the object correspondingly. Do that only if the object is not unlinked.

```
if ((ma->ma_valid & MA_INODE) && (ma->ma_attr.la_nlink == 0))
    RETURN(0);
if (ma->ma_need & MA_STRIPE) {
    rc = mdd_stripe_state(env, obj, ma);
    if (rc)
        RETURN(0);
}
if (ma->ma_need & MA_OBDATTR)
    rc = mdd_lov_attr_get(env, mdd_obj, ma);
```

3. int mdd_lov_attr_get(const struct lu_env *env, struct mdd_object *obj, + struct md_attr *ma)

   Prepares all the needed data to pass the call down to mds and calls mds_lov_attr_get().

```
LASSERT(ma->ma_valid & MA_LOV);
LASSERT(S_ISREG(mdd_object_type(obj)));
OBDO_ALLOC(oa);
if (!oa)
    RETURN(-ENOMEM);
oc = mdo_capa_get(env, obj, NULL, CAPA_OPC_MDS_DEFAULT);
if (IS_ERR(oc))
    oc = NULL;
CDEBUG(D_INODE, "recovering som attributes for "DFID"\n",
       PFID(mdd_object_fid(obj)));
mdd = mdo2mdd(&obj->mod_obj);
obd = mdd2obd_dev(mdd);
rc = mds_lov_attr_get(obd, oa, ma->ma_lmm, ma->ma_lmm_size,
                      ma->ma_cookie_som, oc);
if (rc == 0) {
    obdo_to_la(&ma->ma_attr, oa, oa->o_valid);
    ma->ma_valid |= MA_OBDATTR;
```

```
}
capa_put(oc);
OBDO_FREE(oa);
RETURN(rc);
```

4. static int mdd_fix_attr(const struct lu_env *env, struct mdd_object *obj, struct lu_attr *la, const struct md_attr *ma, int stripe_count)

   The new @stripe_count parameter is added.

   Add LUSTRE_SOM_FL for the inode flags when SOM attributes are updated.

   Make the better assumption this is the SOM attribute update case: LA_SIZE is updated, no MDS_OPEN_OWNEROVERRIDE flag is set, stripe_count is not 0.

```
if (la->la_valid & (LA_SIZE | LA_BLOCKS)) {
    int override = (la->la_valid & MDS_OPEN_OWNEROVERRIDE);
    if (!(override && (uc->mu_fsuid == tmp_la->la_uid)) &&
        !(ma->ma_attr_flags & MDS_PERM_BYPASS)) {
        ...
    } else if (stripe_count && !override) {
        ...
        la->la_valid |= LA_FLAGS;
        la->la_flags = tmp_la->la_flags | LUSTRE_SOM_FL;
}
```

5. static int mdd_attr_set(const struct lu_env *env, struct md_object *obj, const struct md_attr *ma)

   Obtain the MA_LOV when LA_SIZE is updated and pass it to mdd_fix_attr()

```
if (S_ISREG(mdd_object_type(mdd_obj)) &&
    ma->ma_attr.la_valid & (LA_UID | LA_GID | LA_SIZE)) {
    ...
}
...
rc = mdd_fix_attr(env, mdd_obj, la_copy, ma, lmm->lmm_stripe_count);
```

6. static int mdd_notify(struct obd_device *host, struct obd_device *watched, enum obd_notify_event ev, void *owner, void *data)

   The new @data parameter is added. it is passed to all the md_do_upcall() method calls.

   The new event OBD_NOTIFY_CONFIG is handled.

```
case OBD_NOTIFY_SOM_LLOG:
    rc = md_do_upcall(NULL, &mdd->mdd_md_dev, MD_SOM_LLOG, data);
    break;
```

6.3 MDS.

1. Install mds methods to the obd_ops interface.

```
static struct obd_ops mds_obd_ops = {
    ...
    .o_llog_connect    = mds_llog_connect,
};
```

2. int mds_llog_connect(struct obd_export *exp, struct llogd_conn_body *body, struct obd_uuid *uuid)

   The MDS level LLOG_ORIGIN_CONNECT handler on the llog replicator side. Passes the call to LOV.

```
return obd_llog_connect(exp->exp_obd->u.mds.mds_osc_exp, body, uuid);
```

3. int mds_lov_attr_get(struct obd_device *obd, struct obdo *oa, struct lov_mds_md *lmm, int lmm_size, struct llog_cookie *logcookies, struct obd_capa *oc, __u64 ioepoch)

   Similar to ll_inode_getattr(), prepares all the needed data to call obd_getattr().

   Note: the server side lock over obd_getattr() is not needed as all the OST are supposed to be llog-active by this time and therefore data under all the locks on these OSTs are synched as well.

```
LASSERT(lmm);
rc = obd_unpackmd(mds->mds_osc_exp, &oinfo.oi_md, lmm, lmm_size);
if (rc < 0)
    RETURN(rc);
rc = obd_checkmd(mds->mds_osc_exp, obd->obd_self_export, oinfo.oi_md);
if (rc)
    GOTO(out, rc);
oinfo.oi_oa = oa;
oinfo.oi_oa->o_id = oinfo.oi_md->lsm_object_id;
oinfo.oi_oa->o_gr = oinfo.oi_md->lsm_object_gr;
oinfo.oi_oa->o_mode = S_IFREG;
oinfo.oi_oa->o_ioepoch = ioepoch;
oinfo.oi_oa->o_valid = OBD_MD_FLID | OBD_MD_FLTYPE |
                       OBD_MD_FLSIZE | OBD_MD_FLBLOCKS |
                       OBD_MD_FLBLKSZ | OBD_MD_FLATIME |
                       OBD_MD_FLMTIME | OBD_MD_FLCTIME |
                       OBD_MD_FLGROUP | OBD_MD_FLEPOCH;
oinfo.oi_capa = oc;
oinfo.oi_cookies = logcookies;
set = ptlrpc_prep_set();
if (set == NULL) {
```

```
        CERROR("can't allocate ptlrpc set\n");
        GOTO(out, rc = -ENOMEM);
    }
    rc = obd_getattr_async(mds->mds_osc_exp, &oinfo, set);
    if (rc == 0)
        rc = ptlrpc_set_wait(set);
    ptlrpc_set_destroy(set);
    oinfo.oi_oa->o_valid &= (OBD_MD_FLBLOCKS | OBD_MD_FLBLKSZ |
                             OBD_MD_FLATIME | OBD_MD_FLMTIME |
                             OBD_MD_FLCTIME | OBD_MD_FLSIZE |
                             OBD_MD_FLEPOCH);
    EXIT;
    out:
    obd_free_memmd(mds->mds_osc_exp, &oinfo.oi_md);
    RETURN(rc);
```

6.4 LOV.

1. LOV target flags.

   A flag the the OST llogs are completely obtained and handled on the replicator.

   ```
   struct lov_tgt_desc {
       ...
       ltd_llog_active:1;
   };
   ```

2. Install lov methods to the obd_ops interface.

   ```
   struct obd_ops lov_obd_ops = {
       ...
       .o_llog_connect = lov_llog_connect,
       .o_stripe_state = lov_stripe_state,
   };
   ```

3. int lov_llog_connect(struct obd_export *exp, struct llogd_conn_body *body, struct obd_uuid *uuid)

   The LOV level LLOG_ORIGIN_CONNECT handler on the llog replicator side. It finds the proper OSC by the given @uuid and calls obd_llog_connect() for it.

   ```
   LASSERT(uuid != NULL);
   for (i = 0; i < lov->lov_tgt_count; i++) {
       if (!lov->lov_tgts[i] || !lov->lov_tgts[i]->ltd_active)
           continue;
   ```

```
        if (!obd_uuid_equals(uuid, &lov->lov_tgts[i]->ltd_uuid))
            continue;
        child = lov->lov_tgts[i]->ltd_exp;
        rc = obd_llog_connect(child, body, NULL);
        RETURN(rc);
    }
    RETURN(-ENOENT);
```

4. int lov_stripe_state(struct obd_export *exp, struct lov_stripe_md *lsm)

   Checks that all the OSCs of the given @lsm are active and their llogs are active as well.

   Returns values:

   0 if all OST are active and llogs are also active;

   -EAGAIN, if some OST is not active or has not active llogs;

   -EIO, if some OST is permanently deactivated.

```
    for (i = 0; i < lsm->lsm_stripe_count; i++) {
        int idx = lsm->lsm_oinfo[i].loi_ost_idx;
        if (!lov->lov_tgts[idx] || !lov->lov_tgts[idx]->ltd_activate)
            RETURN(-EIO);
        if (!lov->lov_tgts[idx]->ltd_active ||
            !lov->lov_tgts[idx]->ltd_llog_active)
            RETURN(-EAGAIN);
    }
    RETURN(0);
```

5. static int lov_notify(struct obd_device *obd, struct obd_device *watched, enum obd_notify_event ev, void *data)

   The new notify event OBD_NOTIFY_LLOG is handled with no extra change.

   The new notify event OBD_NOTIFY_LLOG_ACTIVE is handled similar to OBD_NOTIFY_ACTIVE, pass @ev to the lov_set_osc_active:

```
    if (ev == OBD_NOTIFY_ACTIVE || ev == OBD_NOTIFY_INACTIVE ||
        ev == OBD_NOTIFY_LLOG_ACTIVE) {
        ...
        rc = lov_set_osc_active(obd, uuid, ev);
        if (rc) {
            CERROR("%sactivation of %s failed: %d\n",
                   (ev == OBD_NOTIFY_INACTIVE) ? "de" :
                   (ev == OBD_NOTIFY_LLOG_ACTIVE) ? "llog " : "",
                   obd_uuid2str(uuid), rc);
            RETURN(rc);
        }
```

6. static int lov_set_osc_active(struct obd_device *obd, struct obd_uuid *uuid, int activate)

   Handle another activeate request, OBD_NOTIFY_LLOG_ACTIVE:

```
act_str = activate == OBD_NOTIFY_LLOG_ACTIVE ? "llog active" :
          activate == OBD_NOTIFY_ACTIVE ? "active" : "inactive";
CDEBUG(D_INFO, "Searching in lov %p for uuid %s (make %s)\n",
       lov, uuid->uuid, act_str);
...
if (activate == OBD_NOTIFY_LLOG_ACTIVE) {
    if (lov->lov_tgts[i]->ltd_llog_active == 1)
        GOTO(out, rc = 0);
    lov->lov_tgts[i]->ltd_llog_active = 1;
} else {
    if (lov->lov_tgts[i]->ltd_active == activate) {
        CDEBUG(D_INFO, "OSC %s already marked as (%s)!\n",
               uuid->uuid, act_str);
        GOTO(out, rc);
    }
    lov->lov_tgts[i]->ltd_active = activate;
    if (activate)
        lov->desc.ld_active_tgt_count++;
    else
        lov->desc.ld_active_tgt_count--;
    /* remove any old qos penalty */
    lov->lov_tgts[i]->ltd_qos.ltq_penalty = 0;
}
CDEBUG(D_CONFIG, "Marking OSC %s as (%s)\n",
       obd_uuid2str(uuid), act_str);
out:
lov_putref(obd);
RETURN(rc);
```

7. void lov_merge_attrs(struct obdo *tgt, struct obdo *src, obd_flag valid, struct lov_stripe_md *lsm, int stripeno, int *set)

   Count the amount of merged stripes.

```
if (*set) {
    ...
} else {
    ...
}
*set += 1;
```

8. static int common_attr_done(struct lov_request_set *set)

Require all the stripes to be available when getting the attributes for some IO epoch.

```
if ((set->set_oi->oi_oa->o_valid & OBD_MD_FLEPOCH) &&
    (set->set_oi->oi_md->lsm_stripe_count != attrset)) {
    CERROR("Not all the stripes had valid attrs\n");
    rc = -EIO;
}
```

9. int lov_prep_getattr_set(struct obd_export *exp, struct obd_info *oinfo, struct lov_request_set **reqset)

   Require all the stripes to be available when getting the attributes for some IO epoch.

```
if (!lov->lov_tgts[loi->loi_ost_idx] ||
    !lov->lov_tgts[loi->loi_ost_idx]->ltd_active) {
        CDEBUG(D_HA, "lov idx %d inactive\n", loi->loi_ost_idx);                if (
            GOTO(out_set, rc = -EIO);
        continue;
}
```

6.5 OSC

1. Install lov methods to the obd_ops interface.

```
struct obd_ops osc_obd_ops = {
    ...
    .o_llog_connect = osc_llog_connect,
};
```

2. static int osc_llog_connect(struct obd_export *exp, struct llogd_conn_body *body, struct obd_uuid *uuid)

   The OSC level LLOG_ORIGIN_CONNECT handler on the llog replicator side. It finds the llog context and calls its connect method.

```
CDEBUG(D_OTHER, "handle connect for %s: %u/%u/%u\n",
       exp->exp_obd->obd_name,
       (unsigned) body->lgdc_logid.lgl_ogr,
       (unsigned) body->lgdc_logid.lgl_oid,
       (unsigned) body->lgdc_logid.lgl_ogen);
ctxt = llog_get_context(exp->exp_obd, body->lgdc_ctxt_idx);
LASSERTF(ctxt != NULL, "ctxt is not null, ctxt idx %d \n",
         body->lgdc_ctxt_idx);
rc = llog_connect(ctxt, 1, &body->lgdc_logid, &body->lgdc_gen, NULL);
if (rc != 0)
```

```
        CERROR("failed to connect rc %d idx %d\n", rc,
               body->lgdc_ctxt_idx);
    RETURN(rc);
```

3. static int osc_llog_init(struct obd_device *obd, struct obd_llogs *llogs, struct obd_device *tgt, int count, struct llog_catid *catid, struct obd_uuid *uuid)

   Install the recovery callback:

```
    ctxt->llog_cb = osc_recov_log_size_cb;
```

4. static int osc_recov_log_size_cb(struct llog_handle *llh, struct llog_rec_hdr *rec, void *data)

```
    lsc = (struct llog_size_change_rec *)rec;
    ctxt = llh->lgh_ctxt;
    obd = ctxt->loc_obd;
    if (!(llh->lgh_hdr->llh_flags & LLOG_F_IS_PLAIN)) {
        CERROR("log is not plain\n");
        RETURN(-EINVAL);
    }
    if (lsc == NULL) {
        rc = obd_notify_observer(obd, obd, OBD_NOTIFY_LLOG_ACTIVE, NULL);
        RETURN(rc);
    }
    lsom.ls_ctxt = ctxt;
    lsom.ls_cookie.lgc_lgl = llh->lgh_id;
    lsom.ls_cookie.lgc_subsys = LLOG_SIZE_ORIG_CTXT;
    lsom.ls_cookie.lgc_index = rec->lrh_index;
    lsom.ls_fid = lsc->lsc_fid;
    lsom.ls_ioepoch = lsc->lsc_ioepoch;
    rc = obd_notify_observer(obd, obd, OBD_NOTIFY_SOM_LLOG, &lsom);
    RETURN(rc);
```

6.6 OST

1. static int ost_lock_get(struct obd_export *exp, struct obdo *oa, __u64 start, __u64 count, struct lustre_handle *lh, int mode, int flags)

   The former ost_punch_lock_get() method, which is reused for getattr.

```
    struct ldlm_res_id res_id = { .name = { oa->o_id, 0, oa->o_gr, 0} };
    end = start + count;
    LASSERT(!lustre_handle_is_used(lh));
    LASSERT((oa->o_valid & (OBD_MD_FLID | OBD_MD_FLGROUP)) ==
```

```
        (OBD_MD_FLID | OBD_MD_FLGROUP));
if (!(oa->o_valid & OBD_MD_FLFLAGS) ||
    !(oa->o_flags & OBD_FL_SRVLOCK))
    RETURN(0);
CDEBUG(D_INODE, "OST-side extent lock.\n");
if (start != 0)
    flags &= ~LDLM_AST_DISCARD_DATA;
policy.l_extent.start = start & CFS_PAGE_MASK;
if (count == OBD_OBJECT_EOF || end < start)
    policy.l_extent.end = OBD_OBJECT_EOF;
else
    policy.l_extent.end = end | ~CFS_PAGE_MASK;
RETURN(ldlm_cli_enqueue_local(exp->exp_obd->obd_namespace, &res_id,
       LDLM_EXTENT, &policy, mode, &flags,
       ldlm_blocking_ast, ldlm_completion_ast,
       ldlm_glimpse_ast, NULL, 0, NULL, lh));
```

2. static void ost_lock_put(struct obd_export *exp, struct lustre_handle *lh, int mode)

   The former ost_punch_lock_put() method, which is reused for getattr.

   ```
   if (lustre_handle_is_used(lh))
       ldlm_lock_decref(lh, mode);
   ```

3. static int ost_getattr(struct obd_export *exp, struct ptlrpc_request *req)

   If SRVLOCK flag is received, get and put the local lock around the operation.

   ```
   if (body->oa.o_valid & OBD_MD_FLFLAGS &&
       body->oa.o_flags & OBD_FL_SRVLOCK) {
       struct lustre_handle lh = {0,};
       CLASSERT(OST_CONNECT_SUPPORTED & OBD_CONNECT_SOM);
       rc = ost_lock_get(exp, &body->oa, 0, OBD_OBJECT_EOF,
                         &lh, LCK_PR, 0);
       if (rc)
           RETURN(rc);
   }
   ...
   if (body->oa.o_valid & OBD_MD_FLFLAGS &&
       body->oa.o_flags & OBD_FL_SRVLOCK)
       ost_lock_put(exp, &lh, LCK_PR);
   ```

4. static int ost_punch(struct obd_export *exp, struct ptlrpc_request *req, struct obd_trans_info *oti)

   Calls ost_lock_get()/_put() instead of ost_punch_lock_get()/_put().

```
rc = ost_lock_get(exp, oinfo.oi_oa, oinfo.oi_oa->o_size,
                  oinfo.oi_oa->o_blocks, &lh, LCK_PW,
                  LDLM_AST_DISCARD_DATA);
...
ost_lock_put(exp, &lh, LCK_PW);
```

5. static int ost_llog_handle_connect(struct obd_export *exp, struct ptlrpc_request *req)

   The OST handler for LLOG_ORIGIN_CONNECT rpc. In contrast to the previous code, it calls llog_processing by itself.

```
body = lustre_msg_buf(req->rq_reqmsg, 1, sizeof(*body));
rc = obd_llog_connect(exp, body, NULL);
if (rc == 0)
    rc = obd_llog_recovery(exp, body, NULL, NULL, NULL);
RETURN(rc);
```

6. static int __ost_synchronize(void *data)

   The OST synchronization thread, it connects to replicators and therefore initiates the OST llog recovery.

   However, first of all, OST does the full sync of all the locks, otherwise a case when a client is evicted by MDS but not evicted by an OST could be missed, and if this client still has some dirty data, we may miss this attribute change later.

```
exp = (struct obd_export *)data;
ptlrpc_daemonize("ost_synchronize");
/* Cancel all the locks first of all. */         ldlm_revoke_ns_locks(exp->exp_obd-
/* LLOG_ORIGIN_CONNECT back to MDS. */
ctxt = llog_get_context(exp->exp_obd, LLOG_SIZE_ORIG_CTXT);
rc = llog_connect(ctxt, 1, NULL, NULL, NULL);
if (rc != 0) {
    CERROR("%s: failed at llog_origin_connect: %d\n",
           exp->exp_obd->obd_name, rc);
}
RETURN(rc);
```

7. static int ost_synchronize(struct obd_export *export, struct ptlrpc_request *req)

   The OST synchronization handler, it creates a separate thread to connect to replicators and recover there the llogs from OST.

```
conn = lustre_msg_buf(req->rq_reqmsg, 1, sizeof(*conn));
```

```
if (conn->lgdc_logid.lgl_ogr < FILTER_GROUP_MDS0)
    RETURN(0);
class_incref(req->rq_export->exp_obd);
rc = cfs_kernel_thread(__ost_synchronize, req->rq_export,
                       CLONE_VM | CLONE_FILES | CLONE_FS);
if (rc < 0) {
    CERROR("%s: error starting ost_synchronize: %d\n",
           export->exp_obd->obd_name, rc);
    class_decref(export->exp_obd);
} else {
    CDEBUG(D_HA, "%s: ost_synchronize thread=%d\n",
           export->exp_obd->obd_name, rc);
    rc = 0;
}
RETURN(rc);
```

8. int ost_handle(struct ptlrpc_request *req)

   The main coming request handler on OST.

   It now starts the LLOG synchronization with replicators which is MDS in our case in response to the LLOG_ORIGIN_CONNECT requests from MDS.

   There is also the handling of LLOG_ORIGIN_HANDLE_NEXT_BLOCK and LLOG_ORIGIN_HANDLE_PREV_BLOCK requests are added, what is essential for llog recovery.

```
case LLOG_ORIGIN_CONNECT:
    ...
    if (rc == 0)
        ost_synchronize(req->rq_export, req);
case LLOG_ORIGIN_HANDLE_NEXT_BLOCK:
    DEBUG_REQ(D_INODE, req, "llog next block");
    OBD_FAIL_RETURN(OBD_FAIL_OBD_LOGD_NET, 0);
    rc = llog_origin_handle_next_block(req);
    break;
case LLOG_ORIGIN_HANDLE_PREV_BLOCK:
    DEBUG_REQ(D_INODE, req, "llog prev block");
    OBD_FAIL_RETURN(OBD_FAIL_OBD_LOGD_NET, 0);
    rc = llog_origin_handle_prev_block(req);
    break;
```

6.7 FILTER

1. static int filter_llog_init(struct obd_device *obd, struct obd_llogs *llogs, struct obd_device *tgt, int count, struct llog_catid *catid, struct obd_uuid *uuid)

Initialize the llog cookie hash on the llog initialization.

```
rc = filter_llog_cookie_fill(obd, ctxt);
```

2. struct obd_llogs *filter_grab_llog_for_group(struct obd_device *obd, int group, struct obd_export *export, int index)

   New parameter @index is added, it defines the llog ctxt index, used not only for LLOG_MDS_OST_REPL_CTXT but for LLOG_SIZE_ORIG_CTXT as well. The old method calls pass the former one, only the new cases with the the later index are described in the DLD.

```
ctxt = llog_get_context_from_llogs(fglog->llogs, index);
```

3. static int filter_getattr(struct obd_export *exp, struct obd_info *oinfo)

   Get the cookie only if the IO epoch is sent by the client. Return the OBD_MD_FLCOOKIE valid flag to the client only if the cookie exists on OST.

```
if (oinfo->oi_oa->o_valid & OBD_MD_FLEPOCH) {
    if (!filter_get_cookie(dentry->d_inode,
        obdo_logcookie(oinfo->oi_oa), oinfo->oi_oa->o_ioepoch))
        oinfo->oi_oa->o_valid |= OBD_MD_FLCOOKIE;
}
```

4. int filter_get_cookie(struct inode *inode, struct llog_cookie *cookie, __u64 ioepoch)

   Put the cookie to the provided place only if the inode is in the same of smaller IO epoch than the wanted one.

```
rc = -ENOENT;
if (ofd && ofd->ofd_epoch <= ioepoch) {
    memcpy(cookie, &ofd->ofd_cookie, sizeof(*cookie));
    rc = 0;
}
...
RETURN(rc);
```

5. static int filter_cat_add_sz_rec(struct llog_handle *llh, struct inode *inode, struct obdo *oa, struct llog_cookie *logcookie)

   Prepares and adds the size llog record, used to be a part of filter_log_sz_change() before. In addition to the previous code, it stores the obtained mds fid to the llog record.

```
ioepoch = oa->o_ioepoch;
OBD_ALLOC_PTR(lsc);
if (lsc == NULL)
    RETURN(-ENOMEM);
lsc->lsc_hdr.lrh_len = lsc->lsc_tail.lrt_len = sizeof(*lsc);
lsc->lsc_hdr.lrh_type = OST_SZ_REC;
lsc->lsc_ioepoch = oa->o_ioepoch;
lsc->lsc_fid = oa->o_mds_fid;
lsc->lsc_id = oa->o_id;
lsc->lsc_gr = oa->o_gr;
rc = llog_cat_add_rec(llh, &lsc->lsc_hdr, logcookie, NULL);
if (rc != 1)
    CWARN("Failed to llog cookie for epoch=%llu ino=%lu\n",
          oa->o_ioepoch, inode->i_ino);
OBD_FREE_PTR(lsc);
RETURN(rc == 1 ? 0 : rc);
```

6. static int filter_cat_cancel_sz_rec(struct llog_handle *llh, struct inode *inode, __u64 ioepoch, struct llog_cookie *logcookie)

   Cancels the llog record by the given llog cookie, if it is valid. Used to be a part of filter_log_sz_change() before.

```
if (!logcookie->lgc_lgl.lgl_oid)
    RETURN(0);

rc = llog_cat_cancel_records(llh, 1, logcookie);
if (rc)
    CWARN("Failed to cancel cookie for epoch=%llu ino=%lu\n",
          ioepoch, inode->i_ino);
RETURN(0);
```

7. static int filter_log_rehash_size(struct inode *inode, __u64 ioepoch, struct llog_cookie *logcookie)

   Allocates a new ost_filter structure if not yet allocated, unhashes it otherwise, initialize it with new data and hash it back. Used to be a part of filter_log_sz_change() before.

```
ofd = inode->i_filterdata;
if (!ofd) {
    OBD_ALLOC_PTR(ofd);
    if (!ofd)
        RETURN(-ENOMEM);
    INIT_HLIST_NODE(&ofd->ofd_list);
    igrab(inode);
    inode->i_filterdata = ofd;
    ofd->ofd_inode = inode;
```

```
        CDEBUG(D_INFO, "LLog cookie for epoch=%llu ino=%lu",
               ioepoch, inode->i_ino);
    } else {
        hlist_del_init(&ofd->ofd_list);
        llog_cookie_count--;
        CDEBUG(D_INFO, "Replace LLog cookie for epoch=%llu to %llu "
               "ino=%lu\n", ofd->ofd_epoch, ioepoch, inode->i_ino);
    }
    ofd->ofd_cookie = *logcookie;
    ofd->ofd_epoch  = ioepoch;
    bucket = llog_cookie_hash + llog_cookie_hashfunc(&ofd->ofd_cookie);
    hlist_add_head(&ofd->ofd_list, bucket);
    llog_cookie_count++;
    RETURN(0);
```

8. int filter_log_sz_change(struct obd_export *exp, struct obdo *oa, struct llog_cookie *logcookie, struct inode *inode)

   In addition to the previous code, it gets the proper llog context from the group llogs. It also uses filter_cat_add_sz_rec(), filter_cat_cancel_sz_rec(), filter_log_rehash_size() methods now.

```
    if (ioepoch == 0)
        RETURN(0);
    /* XXX: get mds_id through the request to FLD. */
    group = FILTER_GROUP_MDS0 /* + mds_id */;
    llogs = filter_grab_llog_for_group(exp->exp_obd, group,
                                       exp, LLOG_SIZE_ORIG_CTXT);
    if (llogs == NULL) {
        CDEBUG(D_HA, "unknown group "LPU64"!\n", group);
        RETURN(-ENOENT);
    }
    ctxt = llog_get_context_from_llogs(llogs, LLOG_SIZE_ORIG_CTXT);
    ...
    rc = filter_cat_add_sz_rec(ctxt->loc_handle, inode, oa, logcookie);
    if (rc) {
        up(&llog_cookie_sem);
        RETURN(rc);
    }
    old_cookie = ofd->ofd_cookie;
    old_epoch  = ofd->ofd_epoch;
    rc = filter_log_rehash_size(inode, ioepoch, logcookie);
    up(&llog_cookie_sem);
    if (rc)
        RETURN(rc);
    filter_cat_cancel_sz_rec(ctxt->loc_handle, inode,
                             old_epoch, &old_cookie);
```

```
RETURN(0);
```

9. static int filter_llog_cookie_cb(struct llog_handle *llh, struct llog_rec_hdr *rec, void *data)

   The callback for filter_llog_cookie_fill(), it is called for every llog record in the llog. It finds the dentry by the object group/id stored in the llog record, hash this cookie and install it into inode's ost_filterdata. In the case more than one llog record is found for this inode, the older one is removed.

```
lsc = (struct llog_size_change_rec *)rec;
obd = (struct obd_device *)data;
dentry = filter_fid2dentry(obd, NULL, lsc->lsc_gr, lsc->lsc_id);
if (IS_ERR(dentry) || dentry->d_inode == NULL) {
    CWARN("cannot find dentry for the llog record: "
          LPU64" "LPU64"\n", lsc->lsc_id, lsc->lsc_gr);
    if (!IS_ERR(dentry))
        f_dput(dentry);
    RETURN(LLOG_DEL_RECORD);
}
down(&llog_cookie_sem);
ofd = dentry->d_inode->i_filterdata;
if (ofd && ofd->ofd_epoch >= lsc->lsc_ioepoch) {
    if (ofd->ofd_epoch >= lsc->lsc_ioepoch)
        CERROR("Inode %ld has llog records for epoch %llu and "
               "%llu\n", dentry->d_inode->i_ino, ofd->ofd_epoch,
               lsc->lsc_ioepoch);
        up(&llog_cookie_sem);
        GOTO(out, rc = LLOG_DEL_RECORD);
}
cookie.lgc_lgl = llh->lgh_id;
cookie.lgc_subsys = LLOG_SIZE_ORIG_CTXT;
cookie.lgc_index = rec->lrh_index;
old_cookie = ofd->ofd_cookie;
old_epoch  = ofd->ofd_epoch;
rc = filter_log_rehash_size(dentry->d_inode,
                            lsc->lsc_ioepoch, &cookie);
up(&llog_cookie_sem);
if (rc)
    RETURN(rc);
filter_cat_cancel_sz_rec(llh, dentry->d_inode,
                         old_epoch, &old_cookie);
EXIT;
out:
f_dput(dentry);
return rc;
```

10. int filter_llog_cookie_fill(struct obd_device *obd, struct llog_ctxt *ctxt)

    Initialize the llog cookie hash, call the llog_process method with filter_llog_cookie_cb for every llog record handling.

    ```
    rc = llog_process(ctxt->loc_handle,
                      filter_llog_cookie_cb, obd, NULL);
    if (rc == LLOG_PROC_BREAK)
        rc = 0;
    RETURN(0);
    ```

11. int filter_recov_log_mds_ost_cb(struct llog_handle *llh, struct llog_rec_hdr *rec, void *data)

    If @rec is NULL, just return. This handles the case when the llog recovery thread finishes to process the llog.

    ```
    if (rec == NULL)
        RETURN(0);
    ```

6.8 OBDO

1. void obdo_from_inode(struct obdo *dst, struct inode *src, struct lu_fid *fid, obd_flag valid)

    A new parameter @fid is added, packs it into obdo structure if OBD_MD_FLFID is specified.

    ```
    if (valid & OBD_MD_FLFID) {
        ...
        if (fid)
            dst->o_mds_fid = fid;
    };
    ```

    For the liblustre version of obdo_from_inode(), just directly take it from @lli->lli_fid.

2. void obdo_to_la(struct lu_attr *la, struct obdo *src, obd_flag valid)

    Copies the attributes from @src to @la according to @valid flags and sets @la->la_valid flags properly.

    ```
    valid &= src->o_valid;
    if (valid & OBD_MD_FLATIME) {
        la->la_atime = src->o_atime;
        la->la_valid |= LA_ATIME;
    }
    if (valid & OBD_MD_FLMTIME) {
        la->la_mtime = src->o_mtime;
    ```

```
    la->la_valid |= LA_MTIME;
}
if (valid & OBD_MD_FLCTIME) {
    la->la_ctime = src->o_ctime;
    la->la_valid |= LA_CTIME;
}
if (valid & OBD_MD_FLSIZE) {
    la->la_size = src->o_size;
    la->la_valid |= LA_SIZE;
}
if (valid & OBD_MD_FLBLOCKS) {
    la->la_blocks = src->o_blocks;
    la->la_valid |= LA_BLOCKS;
}
if (valid & OBD_MD_FLBLKSZ) {
    la->la_blocks = src->o_blocks;
    la->la_valid |= LA_BLKSIZE;
}
if (valid & OBD_MD_FLTYPE) {
    la->la_mode = (la->la_mode & ~S_IFMT)|(src->o_mode & S_IFMT);
    la->la_valid |= LA_TYPE;
}
if (valid & OBD_MD_FLMODE) {
    la->la_mode = (la->la_mode & S_IFMT)|(src->o_mode & ~S_IFMT);
    la->la_valid |= LA_MODE;
}
if (valid & OBD_MD_FLUID) {
    la->la_uid = src->o_uid;
    la->la_valid |= LA_UID;
}
if (valid & OBD_MD_FLGID) {
    la->la_gid = src->o_gid;
    la->la_valid |= LA_GID;
}
if (valid & OBD_MD_FLFLAGS) {
    la->la_flags = src->o_flags;
    la->la_valid |= LA_FLAGS;
}
```

6.9 LLITE

1. int ll_som_update(struct inode *inode, struct md_op_data *op_data)

   The former ll_sizeonmds_update(). If there is an error occurred on getting attributes from the OSTs, sens zeroed attributes with the MF_SOM_CHANGE

flag to the MDS as an indicator about this error, to let MDS to do the proper recovery.

Pass to ll_inode_getattr() the io epoch the attributes are wanted for.

Take MF_GETATTR_LOCK from the given @op_data and pass it as @sync flag into ll_inode_getattr() to indicate a lock over obd_getattr() is needed.

```
old_flags = op_data->op_flags;
op_data->op_flags = MF_SOM_CHANGE;
if (lli->lli_ioepoch == op_data->op_ioepoch) {
    rc = ll_inode_getattr(inode, oa, op_data->op_cookies,
                          op_data->op_ioepoch,
                          old_flags & MF_GETATTR_LOCK);
    if (rc) {
        oa->o_valid = 0;
        if (rc == -ENOENT)
            CDEBUG(D_INODE, "objid "LPX64" is destroyed\n",
                   lli->lli_smd->lsm_object_id);
        else
            CERROR("inode_getattr failed (%d): unable to "
                   "send a Size-on-MDS attribute update "
                   "for inode %lu/%u\n", rc, inode->i_ino,
                   inode->i_generation);
    } else {
        CDEBUG(D_INODE, "Size-on-MDS update on "DFID"\n",
               PFID(&lli->lli_fid));
    }
    md_from_obdo(op_data, oa, oa->o_valid);
}
```

2. int ll_inode_getattr(struct inode *inode, struct obdo *obdo, struct llog_cookie *cookies, __u64 ioepoch, int sync)

   The new parameter @ioepoch is added. Send it to the OST as an indicator which IO epoch the attributes are needed for.

   If @sync is given, tell OST to take a lock over getattr to fluch all the data under other locks.

   ```
   oinfo->oi_oa->o_ioepoch = ioepoch;
   oinfo->oi_oa->o_valid |= OBD_MD_FLEPOCH;
   if (sync) {
       oinfo.oi_oa->o_valid |= OBD_MD_FLFLAGS;
       oinfo.oi_oa->o_flags |= OBD_FL_SRVLOCK;
   }
   ```

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

   Pass the fid to be packed into obdo.

```
obdo_from_inode(oa, inode, &lli->lli_fid, flags);
```

Similar in ll_iocontrol(), ll_truncate(), ll_prepare_write(), ll_inode_fill_obdo(), ll_ap_update_obdo(), do_bio_filebacked()

6.10 LibLustre

1. int llu_som_update(struct inode *inode, struct md_op_data *op_data)

   If there is an error occurred on getting attributes from the OSTs, sens zeroed attributes with the MF_SOM_CHANGE flag to the MDS as an indicator about this error, to let MDS to do the proper recovery.

   Pass to llu_inode_getattr() the io epoch the attributes are wanted for.

   Take MF_GETATTR_LOCK from the given @op_data and pass it as @sync flag into ll_inode_getattr() to indicate a lock over obd_getattr() is needed.

```
old_flags = op_data->op_flags;
op_data->op_flags = MF_SOM_CHANGE;
if (lli->lli_ioepoch == op_data->op_ioepoch) {
    rc = llu_inode_getattr(inode, &oa, op_data->op_cookies,
                           op_data->ioepoch,
                           old_flags & MF_GETATTR_LOCK);
    if (rc) {
        oa->o_valid = 0;
        if (rc == -ENOENT)
            CDEBUG(D_INODE, "objid "LPX64" is destroyed\n",
                   lli->lli_smd->lsm_object_id);
        else
            CERROR("inode_getattr failed (%d): unable to "
                   "send a Size-on-MDS attribute update "
                   "for inode %lu/%u\n", rc,
                   (long long)llu_i2stat(inode)->st_ino,
                   lli->lli_st_generation);
    } else {
        CDEBUG(D_INODE, "Size-on-MDS update on "DFID"\n",
               PFID(&lli->lli_fid));
    }
    md_from_obdo(op_data, &oa, oa.o_valid);
}
```

2. int llu_inode_getattr(struct inode *inode, struct obdo *obdo, struct llog_cookie *cookies, __u64 ioepoch, int sync)

   The new parameter @ioepoch is added. Send it to the OST as an indicator which IO epoch the attributes are needed for.

If @sync is given, tell OST to take a lock over getattr to fluch all the data under other locks

```
oinfo.oi_oa->o_ioepoch = ioepoch;
oinfo.oi_oa->o_valid |= OBD_MD_FLEPOCH;
if (sync) {
    oinfo.oi_oa->o_valid |= OBD_MD_FLFLAGS;
    oinfo.oi_oa->o_flags |= OBD_FL_SRVLOCK;
}
```

6.11 MDC.

1. static void mdc_close_handle_reply(struct ptlrpc_request *req, struct md_op_data *op_data, int rc)

   It puts the special flag MF_SOM_AU into the original CLOSE or DONE_WRITING request which identifies there will be a SOM attribute uodate sent from the client later. This is used on the MDS sade (mdt_epoch_close()) to reconstruct the return code returned origionally.

   Check if OBD_MD_FLGETATTRLOCK has come with reply, set MF_GETATTR_LOCK into @op_date->op_flags if so – MDS asks to perform som attribute GETATTR under the lock.

```
if (req && rc == -EAGAIN) {
        struct mdt_epoch *epoch =
            lustre_msg_buf(req->rq_reqmsg, REQ_REC_OFF, sizeof(*epoch));
        struct mdt_body  *repbody =
            lustre_msg_buf(req->rq_repmsg, REPLY_REC_OFF, sizeof(*repbody));
        epoch->flags |= MF_SOM_AU;
        if (repbody->valid & OBD_MD_FLGETATTRLOCK)
            op_data->op_flags |= MF_GETATTR_LOCK;
    }
```

1. static int mdt_epoch_close(struct mdt_thread_info *info, struct mdt_object *o)

2. int mdt_done_writing(struct mdt_thread_info *info)

   Once the replay is obtained, call mdc_close_handle_replay() to prepare the request for the futher possible replay.

```
mdc_close_handle_reply(req, op_data, rc);
```

6.12 LLOG

1. The LLOG_ORIGIN_CONNECT rpc format declaration.

```
const struct req_format RQF_LLOG_ORIGIN_CONNECT =
    DEFINE_REQ_FMT0("LLOG_ORIGIN_CONNECT",
    llog_origin_connect, empty);
EXPORT_SYMBOL(RQF_LLOG_ORIGIN_CONNECT);
const struct req_msg_field RMF_LLOG_ORIGIN_CONNECT =
    DEFINE_MSGF("llog_connect", 0, sizeof(struct llogd_conn_body),
                lustre_swab_llogd_conn_body);
EXPORT_SYMBOL(RMF_LLOG_ORIGIN_CONNECT);
static const struct req_format *req_formats[] = {
    ...
    &RQF_LLOG_ORIGIN_CONNECT,
};
```

2. void lustre_swab_llog_rec(struct llog_rec_hdr *rec, struct llog_rec_tail *tail)

   Swab the mds fid properly.

```
case OST_SZ_REC: {
    ...
    lustre_swab_lu_fid(&lsc->lsc_fid);
}
```

3. static int log_process_thread(void *args)

   Call the llog processing callback with the NULL llog record at the end of llog processing to inform the upper layers about the llog processing completion.

```
rc = llog_cat_process(llh, cb, NULL);
if (rc != LLOG_PROC_BREAK)
    CERROR("llog_cat_process failed %d\n", rc);
else
    cb(llh, NULL, NULL);
```

6.13 LDLM.

1. void ldlm_add_bl_work_item(struct ldlm_lock *lock, struct ldlm_lock *new, struct list_head *work_list)

   @new may be NULL.

```
...
if (new && new->l_flags & LDLM_AST_DISCARD_DATA)
```

```
        lock->l_flags |= LDLM_FL_DISCARD_DATA;
    ...
    if (new)
        lock->l_blocking_lock = LDLM_LOCK_GET(new);
```

2. void ldlm_revoke_ns_locks(struct ldlm_namespace *ns)

   Revoke all the granted NS locks.

```
CFS_LIST_HEAD(lock_list);
ldlm_namespace_foreach_res(ns, ldlm_revoke_ns_locks_cb, &lock_list);
/* XXX: should -ERESTART be handled somehow ? */
ldlm_run_bl_ast_work(&lock_list);
```

3. static int ldlm_revoke_ns_locks_cb(struct ldlm_resource *res, void *data)

   Traverse all the locks on the given resource, if traversal within a reource stopped, LDLM_ITER_STOP is returned, what means not a granted lock reached, do not stop the next resource traversal.

```
int rc;
rc = ldlm_resource_foreach(res, ldlm_revoke_res_locks_cb, data);
return rc == LDLM_ITER_STOP ? 0 : rc;
```

4. static int ldlm_revoke_res_locks_cb(struct ldlm_lock *lock, void *data)

   If the given @lock is not granted, stop traversal, otherwise add the lock into the @work_list.

```
struct list_head *work_list = (struct list_head *)data;
if (lock->l_req_mode != lock->l_granted_mode)
    return LDLM_ITER_STOP;
ldlm_add_bl_work_item(lock, NULL, work_list);
return LDLM_ITER_CONTINUE;
```

7 State specification.

The recovery of the SOM attributes on MDS consists of many events that may come in different order and therefore the object may pass to a wide range of states. The following graph describes the object states and switches between them.

7.1 Definitions.

'No llogs' – the MDS has not obtained an OST llog record for this object or the recovery from the obtained llog records is not needed.

'LLogs exist' – the MDS has obtained an OST llog record for this object.

LLOG flag – the flag that indicates the object has SOM llogs on OSTs.

7.2 States.

1. Enter/Exit.

```
mot_ioepoch:      any
mot_epochcount:   any
LLOGS flag:       -
in recovery list: -
in failed list:   -
```

2. IO epoch is closed, llogs exist.

```
mot_ioepoch:     >0
mot_epochcount    0
LLOGS flag:       +
in recovery list: +
in failed list:   -
```

3. IO epoch is closed, recovering.

```
mot_ioepoch:     >0
mot_epochcount    0
LLOGS flag:       +
in recovery list: -
in failed list:   -
```

4. IO epoch is closed, recovery failed.

```
mot_ioepoch:     >0
mot_epochcount    0
LLOGS flag:       +
in recovery list: -
in failed list:   +
```

5. IO epoch is opened, no llogs.

   Impossible. It is always the state 1 or 6.

6. IO epoch is opened, llogs exist.

```
mot_ioepoch:     >0
mot_epochcount   >0
LLOGS flag:       +
in recovery list: -
in failed list:   -
```

7. IO epoch is opened, recovering.

   Impossible. it is always the state 1,3 or 6.

8. IO epoch is opened, recovery failed.

   Impossible. it is always the state 1,4 or 6.

9. Pending SOM attribute update, no llogs.

   Impossible. It is always the state 10.

10. Pending SOM attribute update, llogs exists.

```
mot_ioepoch:     >0
mot_epochcount:   0
LLOGS flag:       +
in recovery list: -
in failed list:   -
```

11. Pending SOM attribute update, recovering.

    Impossible. It is always the state 1,3, or 10.

12. Pending SOM attribute update, recovery failed.

    Impossible. It is always the state 1,4, or 10.

13. SOM attributes are updated, no llogs.

```
mot_ioepoch:     >0
mot_epochcount:   0
LLOGS flag:       -
in recovery list: +
in failed list:   -
```

14. SOM attributes are updated, llogs exists.

    Impossible. It is always the state 1,2 or 13.

15. SOM attributes are updated, recovering.

    Impossible. It is always the state 1,3 or 13.

16. SOM attributes are updated, recovery failed.

    Impossible. It is always the state 1,4 or 13.

17. A new IO epoch is opened, no llogs.

    Impossible. It is always the state 1.

18. A new IO epoch is opened, llogs exists.

    Impossible. It is always the state 1.

19. A new IO epoch is opened, recovering.

    Impossible. It is always the state 1. (the recovery will not apply attributes as IOEpoch number will differ).

20. A new IO epoch is opened, recovery failed.

    Impossible. It is always the state 1.

21. Pending new SOM attribute update, no llogs.

    Impossible. It is always the state 1.

22. Pending new SOM attribute update, llog exists.

    Impossible. It is always the state 1.

23. Pending new SOM attribute update, recovering.

    Impossible. It is always the state 1.

24. Pending new SOM attribute update, recovery failed.

    Impossible. It is always the state 1.

25. New SOM attributes are updated, no llogs.

    Impossible. It is always the state 1.

26. New SOM attributes are updated, llogs exists.

    Impossible. It is always the state 1.

27. New SOM attributes are updated, recovering.

    Impossible. It is always the state 1.

28. New SOM attributes are updated, recovery failed.

    Impossible. It is always the state 1.

7.3 Events.

1-4. OPEN replay, CLOSE replay, DW replay, SETATTR replay.

5-8. OPEN resend, CLOSE resend, DW resend, SETATTR resend.

9-12. OPEN, CLOSE, DW, SETATTR.

13. new OST llog record.

14. all OST llog records are obtained.

15. start the recovery.

16. recovery failed.

17. GETATTR-on-MDS.

7.4 The graph.

State 1. Enter. IO epoch is closed, no llogs.

<- state 2,3,4,6,10,13

-> state 2,6

1.	-> state 6.
2,3,4.	Nothing.
5,6,7,8.	Nothing.
9,10,11,12.	Nothing.
13.	if ioepoch > mot_ioepoch: if no IOEpoch is opened -> state 2; otherwise, -> state 6.
14,15,16,17.	Nothing.

State 2. IO epoch is closed, llogs exist.

<- state 1,4,6,10,13

-> state 1,3,6

1.	-> state 6; update mot_ioepoch if greater;
2,3.	Nothing.
4.	(obsolete ioepoch setattr) skip the update.
5.	-> state 1; set SOM_DIRTY flag.
6,7.	Nothing.
8.	(obsolete ioepoch setattr) skip the update.
9.	-> state 1; set SOM_DIRTY flag.
10,11.	Nothing.
12.	(obsolete ioepoch setattr) skip the update.
13.	update mot_ioepoch if greater;
14.	nothing.
15.	-> state 3.
16,17.	(recovery for obsolete ioepoch) skip the update.

State 3. IO epoch is closed, recovering.

<- state 2

-> state 1,4

1,2,3,4.	Nothing.

5.	-> state 1; set SOM_DIRTY flag.
6,7.	Nothing.
8.	(obsolete ioepoch setattr) skip the update.
9.	-> state 1; set SOM_DIRTY flag.
10,11.	Nothing.
12.	(obsolete ioepoch setattr) skip the update.
13.	if ioepoch greater, update mot_ioepoch.
14,15.	Nothing.
16.	if all the OST are connected -> state 4; otherwise, if object ioepoch == the recovery was done for -> mark SOM_DISABELED on disk -> state 1; otherwise, if object ioepoch > the recovery was done for: nothing.
17.	if object ioepoch == the recovery was done for -> setattr the obtained attributes; else: nothing; -> state 27.

State 4. IO epoch is closed, recovery failed.

<- state 3

-> state 1,2

1,2,3,4.	Nothing.
5.	-> state 1. set SOM_DIRTY flag.
6,7.	Nothing.
8.	(obsolete ioepoch setattr) skip the update.
9.	-> state 1. set SOM_DIRTY flag.
10,11.	Nothing.
12.	(obsolete ioepoch setattr) skip the update.
13.	update mot_ioepoch if greater;
14.	-> state 2.
15,16,17.	Nothing.

State 6. IO epoch is opened, llogs exist.

<- state 1,2,10,13.

-> state 1,2,10.

1. nothing.

2,3. if epoch is not closed -> nothing.

if epoch is closed and rc == 0 -> state 2.

if epoch is closed and rc == -EAGAIN and if ioepoch == mot_ioepoch:

-> state 10, set SOM_DIRTY,

else if epoch is closed and rc == -EAGAIN:

-> state 2.

if SOM update came: skip the update.

4. (obsolete ioepoch setattr) skip the update.

5. Nothing.

6,7. if epoch is not closed -> nothing.

if epoch is closed and the ioepoch == mot_ioepoch:

-> state 1, set SOM_DIRTY, return -EAGAIN,

else, if epoch is closed:

-> state 2, return 0.

if SOM update came: skip the update.

8. (obsolete ioepoch setattr) skip the update.

9. Nothing.

10,11. if epoch is not closed -> nothing.

if epoch is closed and the ioepoch == mot_ioepoch:

-> state 1, set SOM_DIRTY, return -EAGAIN,

else, if epoch is closed:

-> state 2, return 0.

if SOM update came: skip the update.

12. (obsolete ioepoch setattr) skip the update.

13. update mot_ioepoch if greater;

14,15,16,17. Nothing.

State 10. Pending SOM attribute update, llogs exists.

<- state 6

-> state -> 1,2,6,13

1. -> state 6.

2,3. Nothing.

4. if ioepoch == mot_ioepoch, apply attributes, -> state 13.

	else: (obsolete ioepoch setattr) skip the update.
5.	-> state 1.
6,7.	Nothing.
8.	if ioepoch == mot_ioepoch, apply attributes, -> state 1.
	else: (obsolete ioepoch setattr) skip the update.
9.	-> state 1.
10,11.	Nothing.
12.	if ioepoch == mot_ioepoch, apply attributes, -> state 1.
	else (obsolete ioepoch setattr) skip the update.
13.	if ioepoch > mot_ioepoch -> state 2, update mot_ioepoch.
	else nothing.
14,15,16,17.	Nothing.

State 13. SOM attributes are updated, no llogs.

<- state 10

-> state 1,2,6

1.	-> state 6.
2,3.	Nothing.
4.	(obsolete ioepoch setattr) skip the update.
5.	-> state 1.
6,7.	Nothing.
8.	(obsolete ioepoch setattr) skip the update.
9.	-> state 1.
10,11.	Nothing.
12.	(obsolete ioepoch setattr) skip the update.
13.	if ioepoch is greater: update mot_ioepoch, -> state 2.
	else: nothing.
14.	nothing.
15.	-> state 1.
16,17.	Nothing.

(** the states 10 and 13 are an optimisation and they are not supported currently, the object gets into the state 1 instead).

8 Protocol, API, disk format.

1. OBDO structure has changed.

   Some unused fields are used for the MDS fid. The size is not changed.

2. SOM_ENABLED flag in inode attributes on MDS.

   Set to the inode attributes every time the SOM attribute update is applied to the inode.

   Can be dropped only if the recovery of inode SOM attributes fails or an OST is permanantly excluded from the cluster.

3. MD object interface mu_upcall().

   New parameter @data is added.

4. OBD interface.

   (a) o_llog_connect().
   Another parameter @uuid is added.

   (b) o_stripe_state()
   New method.

5. OBD NOTIFY interface onu_upcall()

   A new parameter @data is added.

9 Scalability and Performance.

1. In general, Size-on-MDS is supposed to improve the performance dramatically.

2. MDS-OST inode attribute synchronization is performed after the client replay, it does not block anything, at this time SOM is disabled so no performance improvement until all the OST are connected. Once they are connected only the inodes to be recovered have SOM disabled. Once inode attributes are recovered, SOM is enabled on it.

3. If MDS is trying to failover at the time of OST failure, all the inodes to be recovered will be pinned in the memory until OST gets up or another epoch is opened on it. It may result in the performance drop or even in OOM.

10 Test plan.

Test plan is out of scope of this document and will be elaborated as a separate task.

11 Focus on inspection.

1. OBDO structure.

   This is a wire format structure and its change involves the compatibility issues. Despite the fact, the compatibility is our of scope of this document, we need to ensure the current changes will not conflict with the future compatibility protocol.

2. The LLOG'ing code is pretty complex and the inspection needs to be extremely careful here.

12 Questions and Alternatives.

1. IOEpoch generations.

   The IO epoch sequencer cannot be correctly initialized if a client evicted, to not re-use already used IOEpoch number that may present in OST llogs? We do not want to postpone clients activities until after MDS gets synchronized with OST llogs.

2. SOM consistency scanner.

   It could make sance to periodically scan filesystem and check inode SOM attrbute concistency.