Lustre Log API DLD

11 Nov 2005

Contents

1 Requirements

Key requirements for llog are:

- Clear and efficient facility to update of persistent information on multiple systems
- Llogs should be written transactionally
- Llogs only grow and records are cancelled when a commit on local or another system is completed.
- Empty llog can be truncated
- Logs can be removed, remote callers may not assume that open logs will remain available
- Access to logs should be through stateless APIs that can be invoked remotely
- Logs operations should be scalable

Some of the key requirements of these APIs that defines their design are:

- The API should be usable through methods
- The methods should not reveal if the API is being used locally or invoked remotely
- Access to logs should be through stateless APIs that can be invoked remotely
- Access to logs should go through some kind of authorization/authentication system
- API should be clear and flexible. It should be usable for new llog utilization
- API shouldn't has restrictions related to big numbers, i.e. it should be scalable

2 Functional Specification

The typical use of the logging API is the managment of distributed commit of related persistent updates. Other possible use is fast store/retrieve small chunks of data to/from storage.

2.1 Terminology

Originator - the system(s) performing a transaction. The originator starts the action triggered by some internal conditions or events.

Replicator - other system(s) performing a related persistent update. The replicator becomes involved in the action only when a message arrives from originator.

Initiator - the system setting up the connection to execute remote procedure call.

Receptor - the target of that connection event.

2.2 Llog normal operation

2.2.1 Originator

Originator should perform the action and initiate corresponding updates on replicators.

This task can be done using the algorithm below:

1. Starts action triggered by some internal algorithm;
2. the performed action is written transactionally in a log;
   (a) record is added to the llog by *llog_add*();
   (b) a *log_cookie* is generated;
3. sends the action RPC to the replicators involved, through one of the 2.4 *Cookie sending* method.
   (a) if action needs result from replicators immediately then synchronous RPC is used;
   (b) if updates on replicators can be deffered then asynchronous RPC can be used
4. finishes an action;
5. deletes records from log using *llog_cancel*() when the cancel RPC is received from replicator

2.2.2 Replicator

The key requirement is that the replicators must complete their updates if the originators do, even if the originating systems crash or the replicators roll back. Note that we do not require that the the system remains invariant under rollback of the originator.

Replicator algorithm:

1. Receives the originator message;
2. performs the related action and executes a commit callback for that;
3. answers to the originator with action's result;
4. The callback indicates that . The function *lop_cancel* is responsible for this queuing of the cancellation.
5. when the replicators related action commits:
   (a) the *log_cookie* is put up for cancellation;
   (b) cancellations can be grouped together before sending to the originator. See *2.5* section.

2.3 Llog connections

In order to process llog operations and recovery actions, the originators and replicators use a connection to execute remote procedure calls. The connection is used symmetrically, i.e. the originator and replicator can either be the initiator or the receptor.

The obd device store a pointer to the import to be used for queuing RPC's.

The key feature about connection is *llog_generation:*

- It is unchanged while connection exists;
- it is increased when connection is reestablished

The connection is established/re-established by originator only. This is an usual connection used by other subsystems.

1. The originator establishes a connection to the replicator.
2. Prior to sending this the originator increases its *generation*, and includes the *generation* the connect RPC.
3. The replicator receives a llog connect RPC. The handler first updates the generation then initiates processing of the logs.

2.4 Cookie sending methods

2.4.1 Client-originated

Client initiates operations, e.g. unlink, receves *llog_cookie* with result and send cookie to the replicator - OST. Current scheme is used for unlink recovery currently.

2.4.2 MDS(originator)-originated.

MDS sends *llog_cookie* to the replicator right after adding record to the llog. In that case it is usefull to has 'sending daemon' of some kind maybe. MDS-originated way is used in chown update for quota functionality.

2.4.3 Possible approaches

1. **Combined method**. It is MDS-originated initially but if MDS isn't able to send too many RPCs it can be switched to client-originated schema. Possible way to avoid massive RPC on MDS in MDS-originated scheme.

2.5 The cancellation daemon.

A replicator runs a subsystem responsible for collecting groups of *llog_cookies* and sending them to the originator for cancellation of the origin llog records. This is done as a side effect of committing the replicating transaction on the replicator. There can be several originators, cookie groups for each of them are linked into list.

A key element in the cancellation is to distinguish between old and new cookies. Old cookies are those that have a generation smaller than the current generation, new cookies have the current generation. The generation is present in the *llog_context*, hence it is both on the server and on the client.

The cancellation context is responsible for the queueing of cancel cookies. For each originator it is in one of two states:

1. Accepting cookies for cancellation

2. Dropping cookies for cancellation

The context switches from 1 to 2 if a timeout occurs on the cancellation RPC. It switches from 2 to 1 in two cases:

1. A cookie is presented with an *llog_generation* bigger than the one held in the context

2. The replicator receives a *llog_connect* method (which will also carry a new llog generation)

The *llog_generation* is an increasing sequence of 128 bit integers with highest order bits the boot count of the originator and the lower bits the *obd_connect* between the originator and the replicator. The originator increases its generation just before sending the *llog_connect* call, the replicator increases it just prior to beginning the handling of recovery when receiving an *llog_connect* call.

2.5.1 Cancellation timeouts.

If the replicator times out during cancellation, it will continue to process the transactions with cookies. The cancellation context will drop the cookies.
The timeout will indicate to the system that the connection must be recovered.

2.6 Llog recovery handling

When the replicator recieves an *llog_connect* rpc, it increases the llcd's generation, and then spawns a thread to handle the processing of catalogs for the context. For each of the catalogs it is handling, it fetches the catalog's *logid* through an *obd_get_cat* info call. When it has received the catalog *logid*, the replicator calls sync and proceeds with *llog_cat_process*

- It only processes records in logs from previous log connection generations.
- The catalog processing repeats operations that should have been performed by the initiator earlier
  - The replicator must be able to distinguish:

    Done: If the operation already took place. If so it queues a commit cancellation cookie which will cancel the log record which it found in the catalog?s log that is being processed. Because sync was called there is no question that this cancellation is for a committed replicating action.

    Not done: The operation was not performed, the replicator performs the action, as it usually does, and queues a commit cookie to initiate cancellation of the log record.
- When log processing completes, an obd-method is called to indicate to the system that logs have been fully processed. In the case of size recovery, this means that the MDS can resume caching file sizes and guarantee their correctness.

2.6.1 Log removal failure.

If an originator crashes during log removal, the log entries may re-appear after recovery. It is important that the removal of a log from a catalog and the removal of the log file are atomic and idempotent. Upon re-connection, the replicator will again process the log.

2.6.2 File size recovery.

The recovery of orphan deletion is adequately described by 2.6.1. In the case of file size recovery, things are more complicated.

2.7 Llog internals

1. Llog objects can be identified in two ways

   (a) Through a name - The interpretation of the name is upto the driver servicing the call. Typical examples of named llogs are files identified by a path name, text versions of the UUIDs, profile names.

   (b) Through an object identifier or llog-id identifier - A directory of llogs which can lookup a name to get an id can provide translation from naming system to an id based system. In our implementation, we use a file system directory to provide this catalog function.

2. Logs only contain records

3. Records in the logs have the following structure:

   - *llog_rec_hdr* - a header, indicating the index , length and type. The header is 16 bytes long
   - Body which is opaque, 32-bit aligned blob
   - *llog_rec_tail* - length and index of recors for walking backwards, it is 8 byte long

4. The first record in every log is a 8K long *llog_log_hdr*. The body of this record contains:

   - a bitmap of records that have been allocated; bit 0 is set immediately because the header itself occupies it
   - A collection of log records behind the header

5. Records can be accessed by :

   - iterating through a specific log
   - providing a *llog_cookie*, which contains the struct *llog_logid* of the log and the record number in the log file where the record resides.

6. Llog catalog functionality is used to store data about many llogs. A catalog of llogs is held at the top level and provides access to the llogs via *llogid.*

   - A catalog API is provided which exploits the lower lustre log API
   - Catalog entries are log entries in the catalog log which contain the log id of the log file concerned.

2.8 Fundamental data structures

2.8.1 Llog ID

Llog ID is used to get access to the llog as well as the using of name.

```
\begin{lstlisting}
struct llog_logid {
__u64 lgl_oid;
__u64 lgl_ogr;
__u32 lgl_ogen;
} __attribute__((packed));
\end{lstlisting}
```

2.8.2 Logging contexts

Each obd device has an array of logging contexts, *struct llog_ctxt:*

```
\begin{lstlisting}
struct llog_ctxt {
int loc_idx;
struct llog_gen loc_gen;
struct obd_device *loc_obd;
struct obd_export *loc_exp;
struct obd_import *loc_imp;
struct llog_operations *loc_logops;
struct llog_handle *loc_handle;
struct llog_canceld_ctxt *loc_llcd;
struct semaphore loc_sem;
void *llog_proc_cb;
};
\end{lstlisting}
```

The contexts contain:

1. *struct llog_gen loc_gen* - the generation of the logs. This is a 128 bit integer consisting of the mount count of the origianating device and the connection count to the replicators.
2. *struct llog_handle *loc handle* - a handle to an open log ()
3. *struct llog_canceld_ctxt *loc_llcd* - a pointer to the logging commit daemon ()
4. *struct obd_device *loc_obd* - a pointer to the containing obd ()
5. *struct obd_export *loc_exp* - an export to the storage obd for the logs ()
6. *struct llog_operations *loc_logops* - a table of llog methods from General API (2.9)

2.8.3 Llog handle

Llog handle is in-memory descriptor for a log object or log catalog

```
    \begin{lstlisting}
    struct llog_handle {
    struct rw_semaphore lgh_lock;
    struct llog_logid lgh_id; /* id of this log */
    struct llog_log_hdr *lgh_hdr;
    struct file *lgh_file;
    int lgh_last_idx;
    struct llog_ctxt *lgh_ctxt;
    union {
    struct plain_handle_data phd;
    struct cat_handle_data chd;
    } u;
    };
    \end{lstlisting}
```

2.8.4 Llog cookie

```
\begin{lstlisting}
    struct llog_cookie {
    struct llog_logid lgc_lgl;
    __u32 lgc_subsys;
    __u32 lgc_index;
    __u32 lgc_padding;
    } __attribute__((packed));
    \end{lstlisting}
```

2.9 General API

General API describes operations with llog that can be done on any Lustre level. It consider the llog as some abstration without knowing low-level details. All these methods are mapped to real group of method for each OBD with special method table in *llog_context*. Therefore the General API is high-level API for llog functionality.

2.9.1 lop_setup()

Prototype:
\lstinline|int lop_setup(struct obd_device *, int, struct obd_device *, int, struct llog_logid *, struct llog_operations *);|

Parameters:

struct obd_device *obd - current OBD;

int index - index in OBD array of llogs;

struct obd_device *disk_obd - OBD where llog is stored actually;

int count - not used, always 1;

struct llog_logid *logid - llog id;

struct llog_operations *op - list of operations for current OBD

Return Values:

Description:

Initialize llog context for llog with **index**.

2.9.2 lop_cleanup()

Prototype:
\lstinline|int lop_cleanup(struct llog_ctxt *);|

Parameters:

struct llog_ctxt *ctxt - llog context.

Return Values: 0 or error code.

Description:

Clean llog context structure.

2.9.3 lop_create()

Prototype:
\lstinline|int lop_create(struct llog_ctxt *, struct llog_handle **, struct llog_logid *, char *);|

Parameters:

> **struct llog_ctxt * ctxt:** llog context;
>
> **struct llog_handle **handle:** pointer to the resulting llog_handle;
>
> **struct llog_logid *logid:** optional *logid*;
>
> **char *name:** optional name for the llog

Return Values: return code 0 and filled *llog_handle* in case of success or error code otherwise

Description:

> If the log id is not null, open an existing log with this ID. If the name is not NULL, open or create a log with that name. Otherwise open a nameless log. The object id of the log is stored in the handle upon success of opening or creation.

2.9.4 lop_close()

Prototype:
\lstinline|int lop_close(struct llog_handle *);|

Parameters:

> **struct llog_handle *handle** - the already opened *llog_handle.*

Return Values: 0 if successfull or error code otherwise

Description:

> Close the log and free the handle. Remove the handle from the catalog's list of open handles. The log may be zapped if special flag is set

2.9.5 lop_destroy()

Prototype:
\lstinline|int lop_destroy(struct llog_handle *);|

Parameters:

> **struct llog_handle *handle** - opened *llog_handle*

Return Values: 0 if there are no errors or error code otherwise.

Description:

> Close the *handle* and destroy the llog.

2.9.6 lop_read_header()

Prototype:
\lstinline|int *lop_read_header(struct llog_handle *);|

Parameters:

struct llog_handle *handle - opened *llog_handle*;

Return Values: error code in case of failure or zero otherwise.

Description:

Read the header of the llog into the handle and also read the last *rec_tail* in the llog to find the last index that was used in the llog.

2.9.7 lop_add()

Prototype:
\lstinline|int lop_add(struct llog_ctxt *, struct llog_rec_hdr *, struct lov_stripe_md *, struct llog_cookie *, int, llog_fill_rec_cb_t);|

Parameters:

struct llog_ctxt *ctxt - llog context;

struct llog_rec_hdr *rec - record header;

struct lov_stripe_md *lsm - information about lov stripes;

struct llog_cookie *logcookies - array of *llog_cookie;*

int numcookies - number of cookies in **logcookie** array;

llog_fill_rec_cb_t - record filling function

Return Values: error code if failures occur. If successfull then return 0 and filled *llog_cookies* array

Description:

add new records to the llog. This function is used when several replicators are updated, so several llog records should be added with returning several *llog_cookie* structures

2.9.8 lop_cancel()

Prototype:
\lstinline|int lop_cancel(struct llog_ctxt *, struct lov_stripe_md *, int, struct llog_cookie *, int);|

Parameters:

struct llog_ctxt *ctxt - lloging context;

struct lov_stripe_md *lsm - information about lov stripes;

int count - number of *llog_cookie* to cancel;

struct llog_cookie *cookies - array of *llog_cookie;*

int flags - flags to show should be cancels sent now to the originator or not.

Return Values: error code in case of failures or zero otherwise.

Description:

For each cookie in the cookie array, function choose correct llog using the stripe information, the log in-use bit is cleared and either:

- Mark it free in the catalog header and delete it if its empty
- Just write out the log header if the log is not empty

The cookies maybe in different llogs, so we need to get new llogs each time.

2.9.9 lop_write_rec()

Prototype:
\lstinline|int lop_write_rec(struct llog_handle *, struct llog_rec_hdr *, struct llog_cookie *, int, void *, int);|

Parameters:

struct llog_handle *loghandle - opened *llog_handle;*

struct llog_rec_hdr *rec - record header;

struct llog_cookie *logcookies - array of *llog_cookie;*

int numcookies - number of *llog_cookie* in array;

void *buf - record body;

int idx - record index or -1 if records is just *appended*;

Return Values: *llog_cookie* and return code 0 or error code if any error occured.

Description:

Appens or overwrite a record in the log. If *buf* is NULL, the record is complete. If *buf* is not NULL, it is inserted in the middle. Records are multiple of 128bits in size and have a header and tail. Write the cookie for the entry into the cookie pointer.

2.9.10 lop_next_block()

Prototype:
\lstinline|int lop_next_block(struct llog_handle *, int *, int, u64 *, void *);|

Parameters:

struct llog_handle *loghandle - opened *llog_handle;*

int *cur_idx - current index;

int next_idx - next index;

__u64 *cur_offset - offset of **next_idx**;

void *buf - buffer for data;

int_len - length of data.

Return Values:

- sets:
  - *cur_offset* to the furthest point read in the log file
  - *cur_idx* to the log index preceeding *cur_offset*
- returns -EIO/-EINVAL on error

Description:

Index *curr_idx* is in the block at **offset.* Set **offset* to the block offset of recort *next_idx.* Copy *len* bytes from the start of that block into the buffer *buf.*

2.9.11 lop_sync()

Prototype:
\lstinline|int lop_sync(struct llog_ctxt *, struct obd_export *);|

Parameters:

struct llog_ctxt *ctxt - llog context;

struct obd_export *exp - OBD export for sync.

Return Values: 0 or error code

Description:

Flush cached cancels. If reverse import is disconnected, put corresponding canceld_context.

2.9.12 lop_connect()

Prototype:
\lstinline|int lop_connect(struct llog_ctxt *, int, struct llog_logid *, struct llog_gen *, struct obd_uuid *);|

Parameters:

> **struct llog_ctxt *ctxt** - llog context;
> **int count** - count (not used currently);
> **struct llog_logid *logid** - llog id;
> **struct llog_gen *gen** - llog generation;
> **struct obd_uuid *uuid** - OBD UUID.

Return Values: 0 or error code

Description:

> Set connection with replicator. If connection is re-established then the new *llog_generation* is sent to the replicator.

2.10 Catalog API

Llog catalog is a functionality over plain llogs that allow to create llog abstaraction over several physical llogs. Llog catalog has two major uses in Lustre:

1. it is used instead of simple plain llog and can grow over size of plain llog by creating new one seamlessy
2. it can be used as storage for multiple llogs, organizing them and providing access to them by id instead of names.

2.10.1 llog_cat_initialize()

Prototype:
\lstinline|int llog_cat_initialize(struct obd_device *, int)|;

Parameters:

> **struct obd_device *obd** - OBD for llog catalog;
> **int count** - number of sub-llogs.

Return Values: 0 or error code

Description:

> There is a simple master function llog cat initialize for catalog setup that uses and array of object id?s stored on the storage obd of the logging. The logids are stored in an array form and given to the llogging contexts during the lop setup calls made by llog init. It uses support from lvfs to read and write the catalog entries and create or remove them.

2.10.2 int llog_cat_add_rec()

Prototype:
\lstinline|int llog_cat_add_rec(struct llog_handle *cathandle, struct llog_rec_hdr *rec, struct llog_cookie *reccookie, void *buf);|

Parameters:

struct llog_handle *cathandle - current catalog handle;

struct llog_rec_hdr *rec - record header;

struct llog_cookie *reccookie - llog cookie to be returned as result of operation;

void *buf - record body.

Return Values: - error code or zero with *llog_cookie* structure filled.

Description:

Adds new record in current sub-llog. If it is empty, then create new llog.

2.10.3 int llog_cat_put()

Prototype:
\lstinline|int llog_cat_put(struct llog_handle *);|

Parameters:

struct llog_handle *cathandle - llog catalog handle.

Return Values: 0 or error code

Description:

Call *llog_close()* for each plain llog in the catalog and close llog catalog itself.

2.10.4 int llog_cat_cancel_records()

Prototype:
\lstinline|int llog_cat_cancel_records(struct llog_handle *, int, struct llog_cookie *);|

Parameters:

struct llog_handle *cathandle - llog catalog handle;

int count - number of llog cookies in the cookie array;

struct llog_cookie *cookies - array of llog cookies for cancellation.

Return Values: 0 or error code

Description:

Call *llog_cancel_records()* for each llog cookie in array. If sub-llog was destroed, it will be deleted from llog catalog also.

2.10.5 int llog_cat_process()

Prototype:
\lstinline|int llog_cat_process(struct llog_handle *, llog_cb_t, void *);|

Parameters:

- **struct llog_handle *cat_llh** - llog catalog handler;
- **llog_cb_t cb** - llog processing function;
- **void *data** - data for llog processing function.

Return Values: 0 or error code.

Description:

Iterate through all sub-llogs and call *llog_process*() for each of them.

2.11 OBD API

Each OBD supports array of llog contexts for various llogs and each llog has own index in that array. Therefore each llog has own context in all related OBD. Key structure in context is *llog_operations* which maps general llog functions to the real one for that context.

2.11.1 obd_llog_init()

Prototype:
\lstinline|int obd_llog_init(struct obd_device *, struct obd_device *, int, struct llog_catid *);|

Parameters:

- **struct obd_device *obd** - current OBD;
- **struct obd_device *disk_obd** - OBD where llog is setup;
- **int count** - number of sub-llogs in llog catalog. Usually it is equal to the number of replicators.
- **struct llog_catid *logid** - array of *llog_logid* structures according with *count*.

Return Values: 0 or error code.

Description:

This obd method initializes the logging subsystem for an current OBD. It sets the methods and propages calls to dependent OBD's.

2.11.2 obd_llog_finish()

Prototype:
\lstinline|int obd_llog_finish(struct obd_device *, int);|

Parameters:

struct obd_device *obd - current OBD;

int count - not used.

Return Values: 0 or error code.

Description:

Current method calls llog cleanup for current OBD.

2.11.3 OBD llog helpers

The obd_llog API has several methods, **setup**, **cleanup**, **add**, **cancel**, as part of the OBD operations. These operations have 2 implementations:

mds_obd_llog_*: simply redirects and uses the method mds_osc_obd, which is normally the LOV running on the MDS to reach the OST's.

lov_obd_llog_*: calls the method on all relevant OSC devices attached to the LOV. A parameter including striping information of the inode is included to determine which OSC's should generate a log record for their replicating OST.

2.11.4 Llog origin OBD methods

While several OBDs have llog methods that are helpers actually, there is the OBD where llog is created on disk. That OBD has special set of methods:

- **llog_obd_origin_setup**;
- **llog_obd_origin_cleanup**;
- **llog_obd_origin_add**;
- **llog_origin_connect**;

2.11.5 Llog replicator OBD methods

The replicator OBD is the one used on *replicator*

- **llog_obd_repl_cancel**
- **llog_obd_repl_sync**
- **llog_repl_connect**

2.12 LVFS API

LVFS llog API is responsible

2.13 Network API

3 Use Cases

3.1 Deletion of files.

Change needs to be replicated from MDS (originator) to OST?s (replicators):

- The OSC's used by the LOV on the MDS act as originator for the change log, using the storage and disk transactions offered by the MDS:
  - OSC's write log records for file unlink events. This is done through an obd api which stacks the MDS on the LOV on the OSC's. Such events are caused by unlink calls, by closing open but unlinked files, by removing orphans (which is recovery from failed closes) and by renaming inodes when they clobber.
  - The OSC's create cookies to be returned to OSTs. These cookies are piggy backed on the replies of unlink, close and rename calls. In the case of removing orphans the cookies are passed to *obd_destroy* calls executed on the MDS.
- OST's act as replicators, they must delete the objects associated with the inode.
  - Remove objects.
  - Pass OSC generated cookies as parameters to *obd_destroy* transactions.
  - Collect cookies in pages for bulk cancellation RPCs to the OSC on MDS.
  - Cancel records on the OSCs on MDS.

3.2 File size changes.

Changes originate on OSTs, these need to be implemented on the MDS

- Upon the first file size change in an I/O epoch on the OST:
  - Writes a new size changes record for new epoch
  - Records the size of the previous epoch in the record
  - Records the object id of the previous epoch in the record
  - It generates a cancellation cookie
- When MDS knows the epoch has ended:
  - It obtains the size at completion of the epoch from client (or exceptionally from the OST)
  - It obtains cancellation cookies for each OST from the client or from the OSTs

  - It postpones starting a new epoch untill the size is known
  - It starts a setattr transaction to store the size
  - When it commits, it cancels the records on the OSTs

3.3 Configuration updates

3.4 RAID1 OST.

The primary is the originator, the secondary is the replicator

- Writes on the primary are accompanied by a change record for an extent

3.5 Llog using for local purposes

Llog can be as fast way to store/retrieve small amount of data locally. Benefits of that way are the following:

- compact placement of records
- simple and fast adding/deleting of records
- complete API

3.5.1 Join-files

Join file functionality uses llog to store lsm of files to join. The llog is used here not for recovery purposes but as tools to fast store-retrieve small chunks of data.

When two files are joined:

1. lsm-s from second file are added to the first one:
   (a) if first file has no llog yet, the new llog is created and lsm is moved from EA to new llog;
   (b) if second file has lsm-llog already then it iterate through llog and move all lsms to the first file llog;
   (c) if second file has only one lsm in EA then lsm is added to first file llog;
2. records in lsm-llog of second file are cancelled.

4 Logic Specification

4.1 Llog on-disk format and structures

On-disk llog structure consists of header *llog_log_hdr* followed by records. Each record must start with *llog_rec_hdr* structure, end with a *llog_rec_tail* and be a multiple of 256 bits in size.

```
\begin{lstlisting}[label=L:llog_log_hdr, caption={struct\ llog\_log\_hdr}]
#define LLOG_CHUNK_SIZE 8192
#define LLOG_HEADER_SIZE (96)
#define LLOG_BITMAP_BYTES (LLOG_CHUNK_SIZE - LLOG_HEADER_SIZE)
struct llog_log_hdr {
struct llog_rec_hdr llh_hdr;
__u64 llh_timestamp;
__u32 llh_count;
__u32 llh_bitmap_offset;
__u32 llh_size;
__u32 llh_flags;
__u32 llh_cat_idx;
/* for a catalog the first plain slot is next to it */
struct obd_uuid llh_tgtuuid;
__u32 llh_reserved[LLOG_HEADER_SIZE/sizeof(__u32) - 23];
__u32 llh_bitmap[LLOG_BITMAP_BYTES/sizeof(__u32)];
struct llog_rec_tail llh_tail;
} __attribute__((packed));
\end{lstlisting}
\begin{lstlisting}
struct llog_rec_hdr {
__u32 lrh_len;
__u32 lrh_index;
__u32 lrh_type;
__u32 padding;
};
\end{lstlisting}
\begin{lstlisting}
struct llog_rec_tail {
__u32 lrt_len;
__u32 lrt_index;
};
\end{lstlisting}
```

4.2 Generic Llog API

Llog methods

4.2.1 llog_connect()

Prototype:

\begin{lstlisting}
int llog_connect(struct llog_ctxt *ctxt,
int count,
struct llog_logid *logid,
struct llog_gen *gen,
struct obd_uuid *uuid);
\end{lstlisting}
The originator and the replicator establish a connection.

1. The logging subsystem on the originator uses the *lop_connect* method to the replicator. The *lop_connect* call sends the logid's of the open catalog from the originator to the replicator.

2. Just prior to sending this the originator context increases its generation, and includes the generation and the logid in the **lop_connect** method, usually calling **llog_orig_connect**.

3. The replicator now receives a llog connect RPC. The handler is the replicators **lop_connect** (usually **llog_repl_connect**). This method first increases the llcd's generation then initiates processing of the logs.

4.3 LVFS Llog API

typedef int (*llog_cb_t)(struct llog_handle *, struct llog_rec_hdr *, void *);
typedef int (*llog_fill_rec_cb_t)(struct llog_rec_hdr *rec, void *data);
extern struct llog_handle *llog_alloc_handle(void);
int llog_init_handle(struct llog_handle *handle, int flags,
struct obd_uuid *uuid);
extern void llog_free_handle(struct llog_handle *handle);
int llog_process(struct llog_handle *loghandle, llog_cb_t cb,
void *data, void *catdata);
extern int llog_cancel_rec(struct llog_handle *loghandle, int index);
extern int llog_close(struct llog_handle *cathandle);

4.4 Llog Catalog API

struct llog_process_data {
void *lpd_data;
llog_cb_t lpd_cb;
};
struct llog_process_cat_data {
int first_idx;
int last_idx;
/* to process catalog across zero record */

```
    };
    int llog_cat_put(struct llog_handle *cathandle);
    int llog_cat_add_rec(struct llog_handle *cathandle, struct llog_rec_hdr
*rec,
    struct llog_cookie *reccookie, void *buf);
    int llog_cat_cancel_records(struct llog_handle *cathandle, int count,
    struct llog_cookie *cookies);
    int llog_cat_process(struct llog_handle *cat_llh, llog_cb_t cb, void *data);
    int llog_cat_set_first_idx(struct llog_handle *cathandle, int index);
```

4.5 Network Llog API

```
int llog_initiator_connect(struct llog_ctxt *ctxt);
    int llog_receptor_accept(struct llog_ctxt *ctxt, struct obd_import *imp);
    int llog_origin_connect(struct llog_ctxt *ctxt, int count,
    struct llog_logid *logid, struct llog_gen *gen,
    struct obd_uuid *uuid);
    int llog_handle_connect(struct ptlrpc_request *req);
```

4.6 OBD Llog API

4.6.1 llog_init.

This obd method initializes the logging subsystem for an obd. It sets the methods and propages calls to dependent obd's.

4.6.2 llog_cat_initialize.

There is a simple master function llog cat initialize for catalog setup that uses and array of object id?s stored on the storage obd of the logging. The logids are stored in an array form and given to the llogging contexts during the lop setup calls made by llog init. It uses support from lvfs to read and write the catalog entries and create or remove them.

4.6.3 OBD llog helpers

The obd_llog api has several methods, setup, cleanup, add, cancel, as part of the OBD operations. These operations have 3 implementations:

mds_obd_llog_*: simply redirects and uses the method mds_osc_obd, which is normally the LOV running on the MDS to reach the OST's.

lov_obd_llog_*: calls the method on all relevant OSC devices attached to the LOV. A parameter including striping information of the inode is included to determine which OSC's should generate a log record for their replicating OST.

4.6.4 obd_llog_setup

(struct obd_device *obd, struct obd_device *disk_obd, int index, int count, struct llog_logid *idarray)

To activate the catalogs for logging and make their headers and file handles available is fairly involved. Each system that requires catalogs manages an array of catalogs. This function is given an array

of logid's and an index. The index pertains to the array of logs used by an originator, the array of logid's is an array with an entry for each osc in the lov stripe descriptor.

4.6.5 obd_llog_cleanup

```
int obd_llog_cleanup(struct obd_device *).
```

Cleans up all initialized catalog handles for a device.

4.6.6 llog_obd_origin_add

```
(struct obd_export *exp, int index, struct llog_rec_hdr *rec, struct lov_stripe_md *
```

Adds a record to the catalog at index index. The lsm is used to identify how to descend an LOV device. The cookies are generated for each record that is added.

4.6.7 llog_obd_repl_cancel

```
(struct obd_device *obd, struct lov_stripe_md *lsm, int count, struct llog_cookie *c
```

Queue the cookies for cancellation. Flags can be 0 or LLC_CANCEL_NOW for immediate cancellation.

4.7 Llog IOCTLs

5 State Specification

5.1 Normal llog operation

5.1.1 Originator

The log record creation produces a *log_cookie.*

- The *log_cookie* is sent to the **replicator**, through one of the *2.4 Cookie sending* method.

- The **replicator** performs the related transaction and executes a commit callback for that. The callback indicates that the *log_cookie* can be put up for cancellation. The function *lop_cancel* is responsible for this queuing of the cancellation.
- Cancellations can be grouped together before sending to the originator. See *2.5 cancellation daemon* section.
- The **originator** cancels the the log records associated with the cookies through the *lop_cancel* method.The log record creation produces a *log_cookie*.
- The *log_cookie* is sent to the **replicator**, through one of the *2.4 Cookie sending* method.
- The **replicator** performs the related transaction and executes a commit callback for that. The callback indicates that the *log_cookie* can be put up for cancellation. The function *lop_cancel* is responsible for this queuing of the cancellation.
- Cancellations can be grouped together before sending to the originator. See *2.5 cancellation daemon* section.
- The **originator** cancels the the log records associated with the cookies through the *lop_cancel* method.

5.1.2 Replicator

5.2 Llog recovery operation