LLog on top of new device stack

Yury Umanets

28.05.2008

Contents

1 Introduction

In this HLD we have to design how Llog will work on top of new metadata stack, that is, MDD, OSD and others. This is required for number of features in uMDS and uOSS components. This is also required for better portability which is key requirement for upcoming Solaris port.

This is required for the following (order is important):

- Get rid of fsfilt and direct Linux VFS calls in llog backend and use osd (as back-store) and mdd (as namespace provider), which in turn may use some low layer API such as DMU or ldiskfs. That is, portability, to allow same code work in userspace, Linux and Solaris kernels;
- Get rid of obsolete obd interface used in llog and make it work on top of new stack;
- More clean layering, more clear code in llog.

1.1 Used terms

The following terms are used in this design. Please read to avoid confusion.

device stack - list of devices (or device layers) connected to each other in specific order to allow passing control from one layer to another layer. Layers are connected in the order specific for the operations executed on this particular stack. Note that in this HLD we use this term for usual device stacks, used for regular FS operations as well as for device stacks needed for carrying control flow for llog operations. That is, even stacks built for llog called here *device stack*;

device layer - part of device stack, providing specific functionality which is logically separated from the functionality provided by other layers;

log object - object representing one llog and providing interface for accessing it.

2 Requirements

The following requirements should be covered in this work:

- Implement new llog framework based on lu_device framework with imlementing all store access via OSD layer;
- Make sure that generic llog code does not use any system specific APIs (like Linux VFS) and rather delegates all work to lower layers;
- Make sure that mountconf works vis new device stack, that is, does not use fsfilt, direct VFS calls or structures.

3 Functional specification

3.1 Using new stack devices

New llog framework should be implemented as an independent module working on device stack. Main logic should be concentrated in this module and all layers in the stack should be considered not more than **service providers**. In most obvious cases, some of these layers, such as MDD, may be namespace access service provider and some others, like OSD - data storage provider. In more subtle cases, when such layers as MDC used, they may be counted as network service provider or in case with LOV - striping abstraction provider.

Using this paradigm, llog core code is implementing main logic about llog processing, adding new entries, cancels, etc., and calls some API functions, which are really implemented in lower layers and this implementation is really depend on what layer does and basically very simple.

The problematic aspect of working on top of new device stack is that, for implementing all llog functionality on client and server, we need to create new "device stacks" which usually were not existing or were not used for control flow.

Example of non existing device stack: llog using in MGS. What we need is the following: MGS->MDD->OSD. MGS is user of llog framework here. Next, it needs to work on top of MDD as namespace provider because it needs to open llog files using filename (short or full) and this means that we need filename->fid translation. And it uses OSD as data storage provider. This device stack is not existing and must be built in startup.

Example of using non usual device stack: llog using for unlink log in MDS. The following device stack is needed here: MDD->LOV->[OSC1-OSCN]->OSD. MDT is user of llog functionality here as it adds and removes unlink records. LOV is needed here as it is striping abstraction device and will add striping knowledge to whole chain because we need to store unlink records separately for all OSTs. OSC is representation of single OST here and called by LOV according to used order. And next, OSD is called because OSC uses it to store unlink records for its OST.

Why this is problematic aspect is because of the following:

1. We need to be able to build these new device stacks, that is, which layer or what module should do this. On behalf of what operation this should be done, etc;
2. Many of devices supposed to be used for llog operations are not lu_device aware and need to be converted first, which is substantial effort and lots of smaller subtle issues and implementation details.

3.2 Mountconf

There three aspects to note related to mountconf:

1. Both MGC and MGS are going to stop using fsfilt interface. They will be using llog API functions for llog related needs. For things like getting backing device label (used in MGC), new APIs are going to be added to lu_device lower layer APIs.
2. Both MGC and MGS are not lu_device aware and needs to be converted. This is not big deal, just initialization will be changed and few more operation vectors will be added;
3. As they both going to use new llog and they need OSD as storage provider, OSD module should be loaded before them.

3.3 Transactions

Few things to note here:

1. No nested transactions in llog;
2. New transactions API will be used here. No fsfilt related transaction calls going to be used, no fsfilt is going to be used at all;
3. Generic llog code will not use transaction API. Only lower layer devices like OSD, which impelement data store part of llog functionality will do it.

Note that this work does not cover much bigger task “Getting rid of OBD”. It still uses OBD in some limited way to provide required functoinality. For instance, lu_devices for topmost layers are created on behalf of OBD initialization.

4 Use cases

All llogs used in Lustre consist of two parts:

- **Origin llog** - the llog located on the node which modifies llog with some data. This node is that node which maintains llog. *Example:* unlink llog origin is located on MDS as MDS stores there information about unlinked files to be used on OST in recovery time. And opposite, size llog origin is located on OST. OST stores there file size changes to be used later on MDS to check size consistency for files which were not committed before MDS crash;
- **Replay llog** - the llog located on node which is usually interested in llog content but can’t change it directly. For unlink llog, replay llog is located on OST, as that is OST interested to know that all unlinked files which are committed on MDS and not committed on OST do not exist.

4.1 Config log

There are the following use cases for config log:

- Accessing config log from user space via ioctl interface on log orig node. In this case we have the following log devices stack: **MDT->MDD->OSD**. MDT module is needed here to provide ioctl interface to user space utility and forward all commands to MDD->OSD chain. Full stack for this case looks like this: **{ioctl}->MDT->MDD->OSD->{ldiskfs|DMU}**;
- Accessing config log from user space via ioctl interface on nodes others than orig ones. Accessing means here llog parsing and executing configure command related to client. For client, stack looks like the following: **{ioctl}->LLITE->LMV->MDC->{ptlrpc};**
- Configuration processing on nodes others than llog origin node. This uses network operations to send RPC with llog commands and receive RPC with llog blocks to be parsed and executed then on the local node (node other than llog origin node). For this case, device stack is this: **{obdclass}->MGC->{ptlrpc}**;
- Log server running at log origin node. Its purpose is to receive llog operation commands sent from nodes others than llog origin and handle them. This allows all nodes in cluster access log located on llog origin node. This uses the following device stack: **{log_server}->MDT->MDD->OSD->{ldiskfs|DMU};**
- When new node joins cluster, MGS needs to add new record into configuration log. It uses the following device stack: **{mgs_handle}->MGS->MDD->OSD->{ldiskfs|DMU}**.

4.2 MDS_OSS log

There are the following uses cases for this log:

- MDS_OSS origin log. Each MDS maintains log of setattr and unlink operations. This is needed for sending this records later to OSS and let it check if these objects have correct attributes (setattr case) and also to check if OSS has zombie objects which already destroyed according to MDS but still could be alive on OSS due to failure while transactions are not yet committed. This case uses the following device stack: **{mdd_setattr,mdd_unlink}->MDD->LOV->OSC->OSD->{ldiskfs|DMU}**. MDD is used here, as in new metadata stack, MDD is local FS driver and performs all operations like unlink;
- MDS_OSS replay log. OSS needs to access unlink+setattr log on each MDS in cluster. In OSS startup time, it processes MDS_OSS log to check if all object from unlink log are destroyed on OSS. It also checks that all objects from setattr log have correct attributes. To do so OSS uses network llog operations, which

were already mentioned in configuration log. For this case, llog device stack is like the following: **OSD->{ptlrpc}**. OSS also need to sync MDS_OSS log from time to time. This is done in the following cases:

- when current transaction is committed, we are sure, that no zombie OSS objects possible even if OSS fails, we would like to inform origin (MDS) that it can cancel records from current transaction;
- on disconnect, same as above.

4.3 Size log

There are the following use cases for this log:

- OSS logs all size changes performed from the clients. In the case MDS fails, OSS may send these log records to the MDS (at MDS request) to let it check if all objects from the log have correct size which is stored on MDS according to SOM (Size-On-MDS) feature. For doing so OSS uses the following stack: **OSD->{ldiskfs|DMU}**;
- MDS needs to access size log for all OSSes in cluster. In startup time, when OSS is connected to MDS, MDS processes size log to check if all objects from the log have correct sizes. From time to time MDS also want to sync the log when it is guaranteed that current transaction is committed. For doing so MDS uses the following stack: **MDD->LOV->OSC->{ptlrpc};**

4.4 LOVEA log

This is used for Joinfile feature which is not yet implemented in CMD branches so we omit this for the sake of simplicity.

5 Logic specification

5.1 Using new stack devices

Main ideas of new llog implementation are the following:

- Use new lu_device interfaces instead of obsolete obd_device to organize control flow through different layers of MDS or OSS. This means, that all existing or new lu_device-s (aka layers) should implement new set of operations related to llog live cycle on the layer. Whole llog functionality will be scattered over all layers to provide correct layering and maintain logical consistency for llog operations based on specific device layer properties. Examples are the following:

  - MDD layer provides local node namespace service;
  - OSD layer provides backing store service;
  - LOV layer introduces striping knowledge to llog operations.

- Introduce new object entity, parallel to lu_object one - log_object. Use the same principles - log_object is layered object. It is represented in a number of layers to provide some service based on the layer specifics. log_object layers are not the same as lu_object layers as these objects provide interfaces for completely different operations. All device stack are described in section 4 (Use Cases).

5.2 Building device stacks

There is number of new device stacks, described in section 4 (Use Cases). These device stacks should be built somehow and this is how it is going to be implemented.

Configuration log only has record about configuring whole stack. That is, in case of MDS, configuration log only responsible for letting us know that MDS server with these particular parameters should be initialized. In other words, only topmost layer for device stack being initialized is going to be configured using this configuration record. And it is up to this topmost layer how whole stack is going to be built.

Thus, we use the following paradigms about building device stacks:

- Topmost layer of the stack initializes all layers and builds stack (or few stacks) in right order. Example: MDT builds metadata stack for regular FS operations in initialization time. It initializes lu_device for all layers: MDT, CMM, MDD, OSD, MDC and connects them to each other in right order: **MDT->CMM->{MDC|MDD}->OSD**. In same time, MDT also builds stack for MDS_OSS log: **MDT->MDD->LOV->OSC->OSD**.
- For llog needs, for different types of logs we need different stacks and they are initialized again by topmost layers. Clear example is MGS. In init time, it builds own stack used for accessing configuration log: **MGS->MDD->OSD.**

What build means is the following:

- In init time, topmost layer initializes all device layer instances and connects them to layers list in right order. The following operations are executed for each layer from stack:
  - Load module. This is usually done with class_get_type() function which makes sure that specific module is loaded in memory;
  - Once module is loaded, new layer (lu_device) gets allocated and initialized;
  - Once we done wit init, we put lu_device into layers list.

- Same principle about finalization. Topmost layer finalizes all layers in the stack when time comes.

All stacks, for regular FS operations as well as for llog operations are equally functional. This means that they all may be used for regular FS operations as well as for llog operations if there is a need. In some cases there is no need to build device stack for llog as existing stack for regular operations may be used. And opposite is true, if we ave any need to run any operation not related to llog on stack built for llog - we will be able to do that.

Once device stack is built, all objects, regardless lu_objects (regular MDS operations object) or log_object (llog operations specific object) may be initialized using on this stack. They all will have same number of layers and layers implementations as number of layers in the stack.

5.3 Sharing devices

Each node has only one instance of each layer for the sake of simplicity and ability to synchronise operations in Lustre code and do not rely on lower layers implementation. This means that same device instances will be participating in a number of stacks or device stacks.

Example: **MDT->MDD->OSD** is used for handling regular FS operations on MDS. In same time, there is llog device stack **MDT->MDD->LOV->OSC->OSD**. As you can see, MDD and OSD device layers are common for both stacks and we need take this into account in device stacks init time and also, in llog device stacks init time.

5.4 Converting devices to lu_device

There are so many new device stacks in compare to what was required for new MDS stack, that we need to convert many of them and make them lu_device aware.

There are the following:

1. MGS - higher level, server side lu_device, similar to MDT. MGS is topmost layer for stack: **MGS->MDD->OSD->{ldiskfs|DMU};**
2. MGC - client side network layer used for accessing cluster configuration and configuring client nodes;
3. LOV - will be used for MDS_OSS log as the layer aware about data striping;
4. OSC - will be used for MDS_OSS log as one OST server representative.

5.5 Using OSD

OSD is lowest layer in all stacks which want to store something. And llog is not exception here. Number of llog stacks use OSD. Good thing is that, OSD already has:

- Set of functions which implement backing store access functionality with care for proper locking, etc. Such as read, write;
- Transactions using new API provided by DMU (in case of using DMU) and provided by ldiskfs (for ldiskfs store);
- Existing lu_device for attaching device specific operations used in llog. This may be used by MGC for getting store label and in other cases like initialization local layer llog part.

All these, enumerated above may be used in new llog implementation which uses OSD as the only one interface for accessing backing store. No more fsfilt calls!

5.6 Mountconf

In order to implement mountconf related requirements we need to make sure that OSD module is loaded and intialized before anything else related to mountconf. The way we're going get this done is the following.

For MDS, MGS loaded and initialized as first module in server side stack. This is done on behalf of mounting server side Lustre FS. Then MGS builds own stack using process described in section 5.2. In this time, module is loaded and lu_device (layer representative) gets intialized for all layers in MGS stack: MDD and OSD. This insures, that OSD is loaded and initialized before any mountconf related call.

6 State management

6.1 Scalability & performance

Scalability should be improved as we start using new devices stack and this means that we get rid of old fashioned recursive obd calls in llog too.

7 Alternatives

In current vision, we suppose that there are two llog types for each log_object. Namely origin llog object and reply llog object. This paradigm is took from former llog implementation. In same time, it does not look good idea to use this as this introduces many

potential forks in the code, on how to behave for reply and orig llogs. It may be implemented in more generic way, that is, without knowledge which llog is reply and which one is orig. This means that both sides, replay and orig are going to be simultaneously reply and orig logs.

Rationale:

1. Simpler implementation for llog generic code;
2. Some llogs want to be orig and replay in same time;
3. Current llog implementation does strictly follow this paradigm. There are llogs which partially implement functionality of both llog types.