Llog over OSD DLD

Yury Umanets

10.12.2007

Contents

1 Functional Specification

This DLD describes LLog over OSD work which is also described in llog-osd-hld.lyx. Main idea of this work is to get rid of llog code which is tied to lvfs and fsfilt interfaces and replace it with using new MDS stack components. There will be logs working via osd which is lower layer in the stack. Said means that we should design log stack which is not necessary should fit mds stack.

1.1 Interface

As we have various log types we should have same number of interfaces, that is, log operations and number of structs representing each log type. These objects are in fact interface ones, they are the following:

```
struct log_object {
        struct lu_device                    *lo_dev;
        struct log_object_header            *lo_header;
        struct log_object_operations        *lo_ops;
        struct list_head                     lo_linkage;
        unsigned long                        lo_flags;
};

struct config_log_object {
        struct log_object                    clo_log;
        struct config_log_object_operations  *clo_ops;
};

struct mds_oss_log_object {
        struct log_object                    mlo_log;
        struct mds_oss_log_object_operations *mlo_ops;
};

struct size_log_object {
        struct log_object                    slo_log;
        struct size_log_object_operations    *slo_ops;
};
```

1.2 Common methods

Log object operations for these interface objects are the following. Generic log_object which contains common methods for all logs.

```
struct log_object_operations {
        int (*lop_init)(const struct lu_env *env,
                        struct log_object *o);
        void (*lop_fini)(const struct lu_env *env,
                         struct log_object *o);
```

```
        void (*lop_free)(const struct lu_env *env,
                         struct log_object *o);

        int (*lop_start)(const struct lu_env *env,
                         struct log_object *o);
        int (*lop_stop)(const struct lu_env *env,
                        struct log_object *o);
        int (*lop_destroy)(const struct lu_env *env,
                           struct log_object *o);

        int (*lop_connect)(const struct lu_env *env,
                           struct log_object *o,
                           struct obd_uuid *uuid);
        int (*lop_disconnect)(const struct lu_env *env,
                              struct log_object *o);
};
```

1.2.1 lop_init() method

Description

This method is called for the following main purposes:

- Initialize log object on its layer, set initial values for object fields, etc;
- Allocate log object for next, bellow layer calling ->ldo_log_alloc() for it;
- Add next layer to list of layers.

Return value

Returns 0 on success and error code otherwise.

1.2.2 lop_fini() method

Description

This method finalizes log object. All external resources and refs should be released. This is called to inform log object that it is going to be freed soon.

Return value

None

1.2.3 lop_free() method

Description

This method frees memory occupied by log object on current layer and naturally is complement to ->ldo_log_alloc()

Return value

None

1.2.4 lop_start() method

Description

This method is called when layered log object is completely allocated and all layers get informed that they start working. This is good time to perform log open or create if not exists yet, header read for config log and similar things for other types of logs.

Return value

Returns 0 on success and error code otherwise.

1.2.5 lop_stop() method

Description

Complement for previous method. This is called when layered object is still fully alive but is going to be freed soon.

Return value

None.

1.2.6 lop_destroy() method

Description

Called to destroy existing log totally.

Return value

Returns 0 on success and error code otherwise.

1.2.7 lop_connect() method

Description

Called to connect remote node from current node. This method is also called on remote node, when handling connection.

Return value

Returns 0 on success and error code otherwise.

1.2.8 lop_disconnect() method

Description

Called to disconnect remote node from current node. This method is also called on remote node, when handling disconnection.

Return value

Returns 0 on success and error code otherwise.

1.3 Config log methods

Config log interface, in addition to common methods contains also methods for adding or deleting records as well as methods for parsing the log using block interface.

```
struct config_log_object_operations {
        int (*lop_add)(const struct lu_env *env,
                       struct log_object *o,
                       struct llog_rec_hdr *hdr,
                       void *rec, int idx);
        int (*lop_del)(const struct lu_env *env,
                       struct log_object *o,
                       int idx);

        int (*lop_next)(const struct lu_env *env,
                        struct log_object *o,
                        int *curr_idx, int next_idx,
                        __u64 *offset,
                        void *buf, int len);
        int (*lop_prev)(const struct lu_env *env,
                        struct log_object *o,
                        int prev_idx, void *buf,
                        int len);
};
```

1.3.1 lop_add() method

Description

This method is called to add new record to config log.

Return value

Returns 0 on success and error code otherwise.

1.3.2 lop_del() method

Description

This method is called to del record from config log.

Return value

Returns 0 on success and error code otherwise.

1.3.3 lop_next() method

Description

This method is called to fetch next log block;

Return value

Returns 0 on success and error code otherwise.

1.3.4 lop_prev() method

Description

This method is called to fetch prev log block;

Return value

Returns 0 on success and error code otherwise.

1.4 MDS_OSS log methods

This is interface for MDS_OSS log which requires ability to add lsm's to the log.

```
struct mds_oss_log_object_operations {
        int (*lop_add)(const struct lu_env *env,
                       struct log_object *o,
                       struct llog_rec_hdr *hdr,
                       struct lov_stripe_md *lsm,
                       struct llog_cookie *cookie,
                       int numcookies);
};
```

1.4.1 lop_add() method

Description

This method is called to add record about unlink or setattr to the log. This includes also object lsm to be able later send it to OSS and let it ability to find local object but lsm attributes;

Return value

Returns 0 on success and error code otherwise.

1.5 Size log methods

Same interface as previous for size log. We keep it separately for the sake of further changes as this is completely different type of log.

```
struct size_log_object_operations {
        int (*lop_add)(const struct lu_env *env,
                       struct log_object *o,
                       struct llog_rec_hdr *hdr,
                       struct lov_stripe_md *lsm,
                       struct llog_cookie *cookie,
                       int numcookies);
};
```

1.5.1 lop_add() method

Description

This method is called to add record about size change on OSS to the log.

Return value

Returns 0 on success and error code otherwise.

1.6 Log stacking

Each log has own presentation on each level of MDS stack and this way forms own log stack. According to this for example MDS_OSS log will have *struct mdd_mds_oss_log_object* as mdd layer MDS_OSS log presentation and so on for all other logs and layers. Log stack is built same way as lu_object stack, this makes it possible for log presentation on current layer call any method for log on lower layers.

1.6.1 Layered log object initialization

Layered log object is initialized in same way as lu_object. That is, there is generic function log_object_alloc() which allocated each layer, sets up layers list and later calls init for all layers in the list. Allocation is done using lu_device methods ->ldo_log_alloc() which has all arguments required for initializing all types of logs. Alternatively there may be own allocation function for each log type.

One important thing which is also done by log_object_alloc() is set up of common log_object header with list of all layers which is later used for calling all methods on all layers. Common means that there is only one lu_object

header instance allocated by topmost layer and all other layers have reference to it, though each layer has own log_object as it contains pointers to methods related to live cycle which are current layer specific and only header is the same for all layers. See section 3.3 for more details on how may allocation look like.

So what we have as a conclusion here is the following:

1. Each log layer is allocated by ->ldo_log_alloc() function of corresponding lu_device. This means that it will allocate the structure specific for current layer of log which is backed out by current layer of lu_device. Say mdd device layer for MDS_OSS log will allocate *struct mdd_mds_oss_log_object.* Note that log_object_alloc() allocates only top layer object and all other layers are allocated step by step by the way of layered log initialization in the way like each layer in its initialization time allocates next layer and adds it to layers list. See section 3.3 for more details on what log object for each layer may exist;

2. Layered log object contains own structures for each layer with own methods set for each layer and only one common structure - reference to log object header. Header contains list of layers and other common information like log name, etc;

3. Each layer may call any other layer (though usually calls bottom layer) using log object common header.

1.6.2 Layered log object finalization

Finalization for layered log object is also done with using common log_object header. Finalization function iterates over all layers and calls finalization function for each of them. See section 3.3 for details of how finalization may look like.

1.6.3 Layered log object functioning

There are defined the following log presentation objects for all layers. Each call to any method goes from up to down of the stack and each layer knows what to do on current layer and what to expect from calling of the lower layer. In some cases there will not be calls down to the stack.

For config log, there are the following layer presentation objects.

```
struct mdt_config_log_object {
        struct config_log_object  clo_obj;
        ...

        /* MDT specific fields for MDT config log */
};

struct osd_config_log_object {
```

```
        struct config_log_object  clo_obj;
        ...

        /* OSD specific fields for OSD config log */
};

struct client_config_log_object {
        struct config_log_object  clo_obj;
        ...
};
```

For MDS_OSS log:

```
struct mdd_mds_oss_log_object {
        struct mds_oss_log_object mlo_obj;
        ...
};

struct lov_mds_oss_log_object {
        struct mds_oss_log_object mlo_obj;
        ...
};

struct osc_mds_oss_log_object {
        struct mds_oss_log_object mlo_obj;
        ...
};

struct osd_mds_oss_log_object {
        struct mds_oss_log_object mlo_obj;
        ...
};
```

For size log:

```
struct osd_size_log_object {
        struct size_log_object    slo_obj;
        ...
};

struct mdd_size_log_object {
        struct size_log_object    slo_obj;
        ...
};

struct lov_size_log_object {
        struct size_log_object    slo_obj;
```

```
        ...
};

struct osc_size_log_object {
        struct size_log_object    slo_obj;
        ...
};
```

Each layer of layered log object has reference to common for all layers log_object header instance. For lov_size_log_object this reference looks like the following:

```
->slo_obj.slo_log.lo_header
```

Header contains list of all layers, and may be used for accessing objects on any layer. Log object common log_object header is allocated and set up in log_object allocation and initialization time which is described in previous section. This means that each layer may find any other layers to call it. In most of cases it needs to call next layer. So that say lov_size_log_object may call add() method for osc_size_log_object and it in turn will forward add() call to lower layer which is osd.

This may look like the following:

```
struct size_log_object *
lov_size_log_child(struct lov_size_log_object *o)
{
        return container_of0(log_object_next(lov2log_obj(o)),
                             struct size_log_object, slo_log);
}

struct lov_size_log_object *
size2lov_size_obj(struct size_log_object *slo)
{
        return container_of0(slo, struct lov_size_log_object,
                             slo_obj);
}

int lov_size_log_add(const struct lu_env *env,
                     struct size_log_object *o,
                     struct lov_stripe_md *lsm, ...)
{
        struct lov_size_log_object *lobj = size2lov_size_obj(o);
        struct size_log_object *child = lov_size_log_child(lobj);
        return child->lop_add(env, child, lsm, ...);
}
```

1.7 Inter-node connections

There is need to connect logs on different nodes. Most obvious example of this is MDS_OSS log. It connects MDS log to OST log n startup. In former solution this is required for the following:

- Exchange log id to find corresponding log instance on remote OST node, setup inter-node relation using log id to be able to find right log instance on remote node for all remote log operations;
- Start special thread running on this log. Purpose of this thread on OST is to parse MDS log using client log operations and make sure that all objects from MDS ORIG log are:
  - destroyed for unlink case;
  - have correct attributes for setattr case.

So here as you can see, though we connect nodes to each other, standard client-server paradigm does not really work as we connect server to client rather. However it does not mean that in new implementation we should follow this way.

All we have to take into account is this:

- Logs on different nodes have to be tied to be able to call methods of each other in client-server fashion;
- In this client-server working style, client often needs to process server log and apply some policy function for each record.

1.8 Using log in mountconf

There is one important using of new logs. This is mountconf. Issue is that, osd should be loaded and initialized before mountconf what is not case right now. This allows mountconf use config log which is osd based on lower layer.

List of things to check is the following:

1. Make sure that OSD module is loaded before mountconf;
2. Make sure that OSD is not configured with mountconf otherwise it would not be possible to use it as mountconf backend.

Read section 3.4 for more details.

1.9 Transactions management

2 Use Cases

No new use cases required, there is special llog sanity which should be passed with limited number of changes (in API mostly) for all log types.

3 Logic Specification

3.1 Overall picture

Overall picture of log functioning looks like the following:

- At startup time all nodes initialize all logs;
- When connections between nodes are established, following should be done:
  - all client logs should be connected to server nodes;
  - all logs on servers try to connect to each other. If some of them do not have connect method implemented, this means that they do not need to connect, but this should be checked in log generic code and if lop_connect() implemented - use it;
- In working time, logs are used locally via layered object or remotely via client log object and its operations which form special set of RPCs with log commands like next block, prev block, etc;
- In shutdown time, logs are disconnected first and then finalized.

3.2 Layered log object initialization

Log allocation function may look like the following.

```
struct log_object *log_object_alloc(const struct lu_env *env,
                                    struct lu_device *d)
{
        struct log_object *scan;
        struct log_object *top;
        struct list_head *layers;
        int clean, result;
        top = d->ld_ops->ldo_log_alloc(env, NULL, name, d);
        if (IS_ERR(top))
                RETURN(top);
        OBD_ALLOC_PTR(top->lo_header);
        if (top->lo_header == NULL) {
                log_object_free(env, top);
                RETURN(ERR_PTR(-ENOMEM));
        }
        log_object_header_init(top->lo_header);
        log_object_add_top(top->lo_header, top);

        layers = &top->lo_header->loh_layers;
        do {
```

```
                clean = 1;
                list_for_each_entry(scan, layers, lo_linkage) {
                        if (scan->lo_flags & LOG_OBJECT_ALLOCATED)
                                continue;
                        clean = 0;
                        scan->lo_header = top->lo_header;
                        result = scan->lo_ops->lop_init(env, scan);
                        if (result != 0) {
                                log_object_free(env, top);
                                RETURN(ERR_PTR(result));
                        }
                        scan->lo_flags |= LOG_OBJECT_ALLOCATED;
                }
        } while (!clean);
        list_for_each_entry_reverse(scan, layers, lo_linkage) {
                if (scan->lo_ops->lop_start != NULL) {
                        result = scan->lo_ops->lop_start(env, scan);
                        if (result != 0) {
                                log_object_free(env, top);
                                RETURN(ERR_PTR(result));
                        }
                }
        }
        RETURN(top);
}
```

As it is seen from proto, it allocates top layer object and calls init for it. Init then adds next layer which is also calls init and so on. When whole object is initialized - lop_start() is called for all layers to let them know that object is ready and log may start working.

3.3 Layered log object finalization

Log object freeing may look like the following:

```
void log_object_free(const struct lu_env *env,
                     struct log_object *o)
{
        struct list_head splice;
        struct log_object *scan;
        list_for_each_entry_reverse(scan,
                                    &o->lo_header->loh_layers,
                                    lo_linkage) {
                if (scan->lo_ops->lop_stop != NULL)
                        scan->lo_ops->lop_stop(env, scan);
        }
```

```
        list_for_each_entry_reverse(scan,
                                    &o->lo_header->loh_layers,
                                    lo_linkage) {
                if (scan->lo_ops->lop_fini != NULL)
                        scan->lo_ops->lop_fini(env, scan);
        }

        INIT_LIST_HEAD(&splice);
        list_splice_init(&o->lo_header->loh_layers, &splice);

        if (o->lo_header != NULL) {
                log_object_header_fini(o->lo_header);
                OBD_FREE_PTR(o->lo_header);
        }

        while (!list_empty(&splice)) {
                o = container_of0(splice.next, struct log_object,
                                  lo_linkage);
                list_del_init(&o->lo_linkage);
                LASSERT(o->lo_ops->lop_free != NULL);
                o->lo_ops->lop_free(env, o);
        }
}
```

3.4 Using log in mountconf

To implement mountconf via osd we need to do the following:

1. Load osd module before mountconf;
2. Add lu_device as device backend to mountconf. All mountconf backing device access should go through this lu_device;
3. Re-implement OSD configuration, read bellow in section 3.4.1;
4. Replace mountconf methods accessing backing store via FSFILT interface with corresponding methods of OSD being called from mountconf using lu_device interface.

3.4.1 OSD configuration changes

Currently mountconf does not work via OSD and rather access backing store using alternative FSFILT interface. Using this approach mountconf access configuration log and performs device stack configuration including OSD as lower layer of the stack. This should be changed as we want to get rid of FSFILT interface in described modules interaction.

To do so we need to do the following:

1. Add lu_device backend to mountconf and change its logic making it work via this lu_device interface;

2. At the beginning, when mountconf requires access the backing store via OSD and it is not yet mounted, it calls *->ldo_process_config(LCFG_SETUP)* which in turn forward the call to *osd_process_config()* to let OSD know what device should be mounted;

3. When OSD is configured this way, mountconf may use it for getting access to log records using config log interface. It also will call *->ldo_process_config(LCFG_SETUP)* for the rest of stack using configuration read via OSD to setup the rest of layers;

4. At shutdown time mountconf calls *->ldo_process_config(LCFG_CLEANUP)* to shut OSD down.

3.5 Transactions management

4 State Specification

4.1 Resources Involved and Their State

The main resource in this work is log objects. We need to support the following invariants:

1. Reference counting. Each log object should have increased refc on the time of any operations done on it. Also all users such as lu_device should take care about this;

2. Initialization and finalization should be balanced in the meaning that there should be simple code and limited number of such calls their number should be balanced.

4.2 Locking

No special locking changes required. All log object fields on all layers should be protected with own locks but this is usual technique.

4.3 Recovery

There is log_recovery_thread() serving some logs interaction. Logic of this thread will not be changed, only implementation.

5 Environment

No changes in protocol or wire. We would like to keep it the same and only change the implementation.