Lustre MGC and Obdclass Deep Dive

Anjus George

HPC Systems Software Engineer

Oak Ridge National Laboratory

May 9th , 2022

ORNL is managed by UT-Battelle LLC for the US Department of Energy

Presentation Outline

MGC	Obdclass
• Introduction to MGC • MGC module initialization • MGC obd operations • mgc_setup() • Operation • Lustre log handling • Log processing in MGC • mgc_precleanup() and mgc_cleanup() • mgc_import_event()	• Introduction to Obdclass • obd_device structure • MGC life cycle • Obd device life cycle • class_attach() • obd_export structure • class_setup() • class_precleanup() and class_cleanup() • Imports and exports • Useful APIs in Obdclass

Introduction to MGC

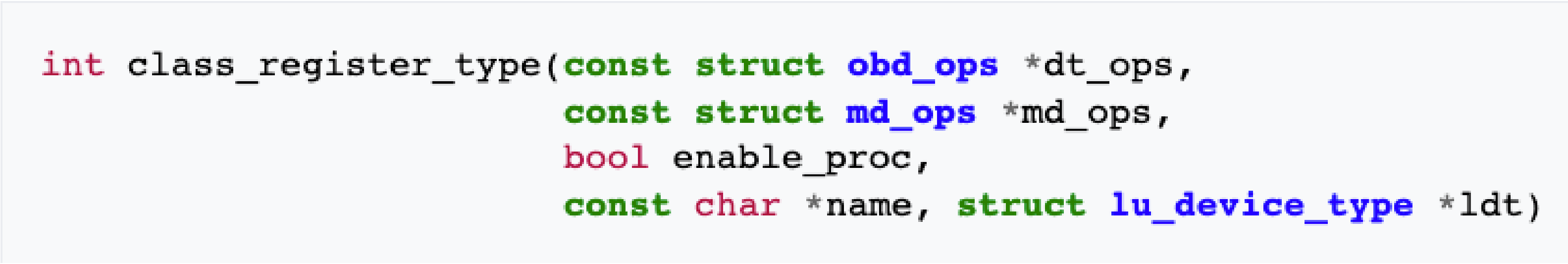
- Lustre client software has 3 components:
  - MGC (Management Client)
  - MDC (Metadata Client)
  - OSC (Object Storage Client)
- MGC acts as interface between virtual file system layer and Management Server (MGS)
- MGS provides configuration information to all components
- Lustre targets register with MGS to provide information to MGS
- Lustre clients contact MGS to retrieve information from it

Introduction to MGC

- Major functionalities of MGC:
  - Lustre log handling
  - Distributed lock management
  - File system setup
- MGC is the first obd device created in Lustre obd device lifecycle
- Obd device in Lustre provides a level of abstraction on Lustre components
  - Generic operations can be applied without knowing the specific device details

MGC Module Initialization

- When MGC kernel module initializes, it registers as an obd device with Lustre using `class_register_type()`

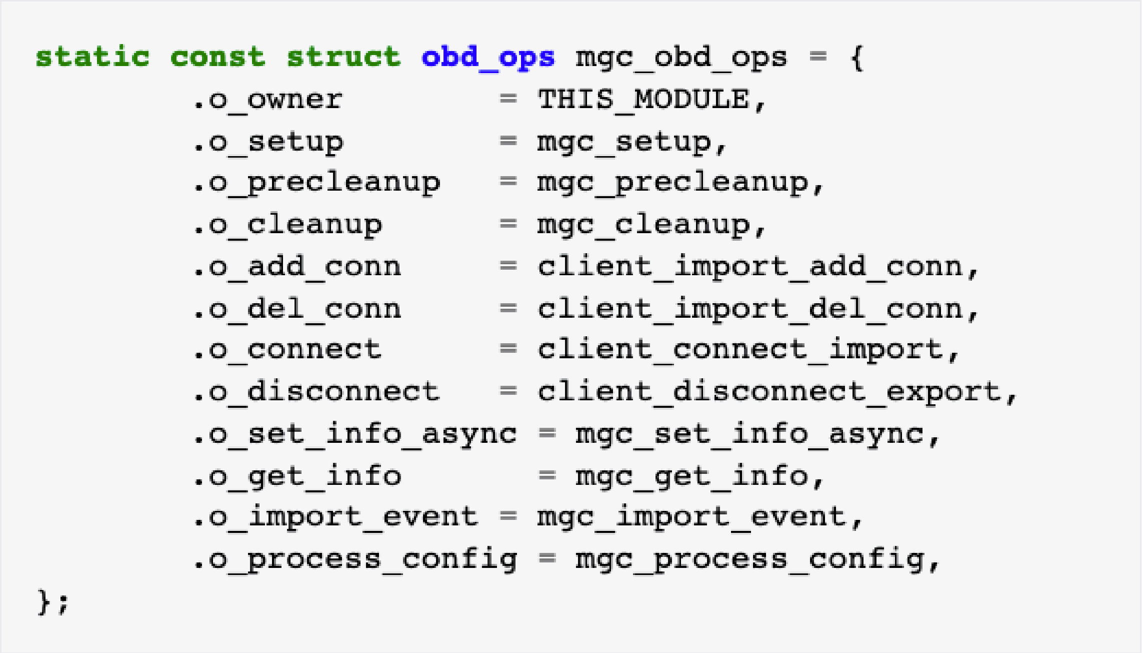


```
int class_register_type(const struct obd_ops *dt_ops,
                        const struct md_ops *md_ops,
                        bool enable_proc,
                        const char *name, struct lu_device_type *ldt)
```

- Obd device data and metadata operations are defined using obd_ops and md_ops structures
- Since MGC deals with metadata, only obd_ops is defined
- `class_register_type()` passes &mgc_obd_ops, and `LUSTRE_MGC_NAME` as its arguments
- `LUSTRE_MGC_NAME` is defined as “`mgc`” in `include/obd.h`

MGC Obd Operations

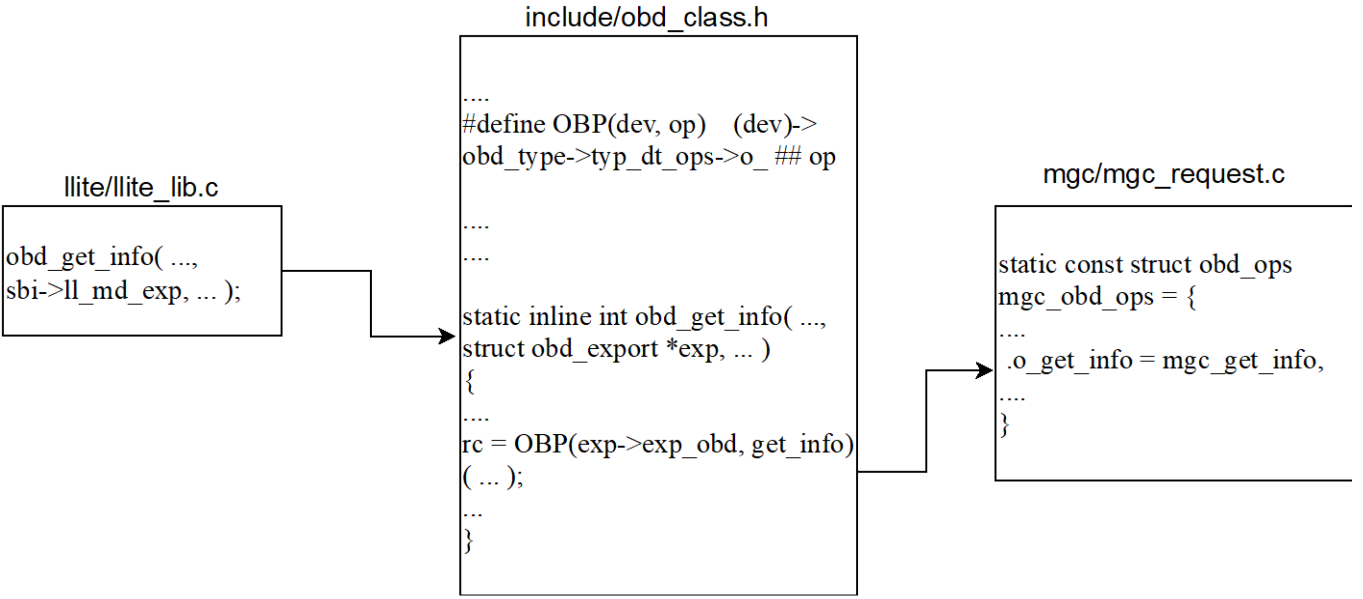
- Defined by mgc_obd_ops structure
- All operations are defined as function pointers
- Provides level of abstraction on Lustre components
- Main operations described here are,
  - mgc_setup()
  - mgc_precleanup()
  - mgc_cleanup()
  - mgc_import_event()
  - mgc_process_config()



```
static const struct obd_ops mgc_obd_ops = {
        .o_owner          = THIS_MODULE,
        .o_setup          = mgc_setup,
        .o_precleanup     = mgc_precleanup,
        .o_cleanup        = mgc_cleanup,
        .o_add_conn       = client_import_add_conn,
        .o_del_conn       = client_import_del_conn,
        .o_connect        = client_connect_import,
        .o_disconnect     = client_disconnect_export,
        .o_set_info_async = mgc_set_info_async,
        .o_get_info        = mgc_get_info,
        .o_import_event = mgc_import_event,
        .o_process_config = mgc_process_config,
};
```

MGC Obd Operations (cont.)

- obd_ops structure can be used to share information between two subsystems
- Shows the example of `mgc_get_info()` from `mgc_obd_ops`
- Llite makes a call to `mgc_get_info()`
- Llite invokes `obd_get_info()` instead of `mgc_get_info()`
- `obd_get_info()` invokes OBP macro by passing `obd_export` structure with `get_info` operation
- OBP concatenates "o" with operation which results in `o_get_info()`



```
obd_get_info( ...,
sbi->ll_md_exp, ... );
```

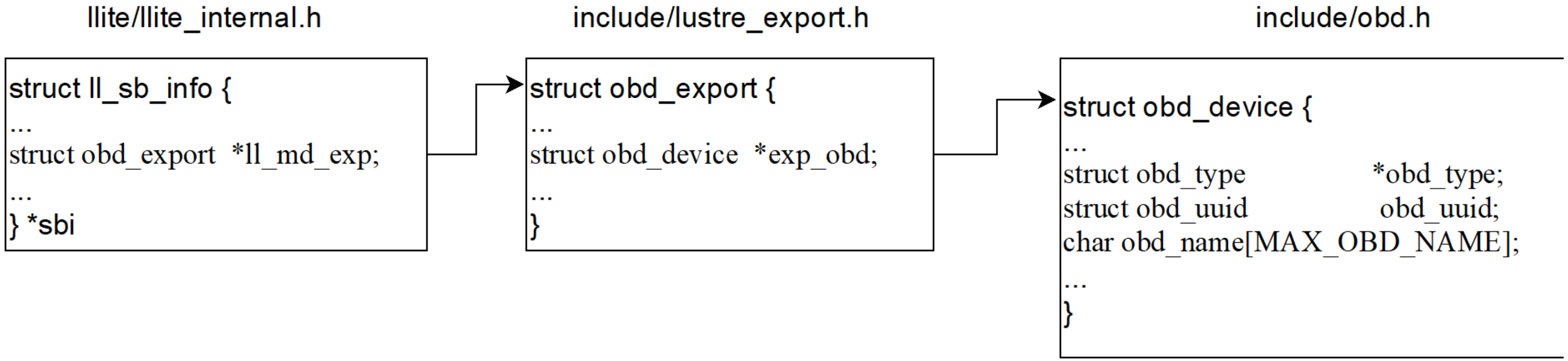
```
....
#define OBP(dev, op)    (dev)->
obd_type->typ_dt_ops->o_ ## op

....
....

static inline int obd_get_info( ...,
struct obd_export *exp, ... )
{
....
rc = OBP(exp->exp_obd, get_info)
( ... );
...
}
```

```
static const struct obd_ops
mgc_obd_ops = {
....
 .o_get_info = mgc_get_info,
....
}
```

Figure 1. Communication between llite and mgc through obdclass[1]

MGC Obd Operations (cont.)

- How does llite make sure to get the correct export for mgc ?
- obd_get_info() has an argument called sbi->ll_md_exp
- sbi is a type of ll_sb_info defined in llite_internal.h
- ll_md_exp field from ll_sb_info is a type of obd_export structure
- obd_export has a field *exp_obd which is an obd_device structure
- obd_connect() retrieves export using the obd_device structure



llite/llite_internal.h

```
struct ll_sb_info {
...
struct obd_export   *ll_md_exp;
...
} *sbi
```

include/lustre_export.h

```
struct obd_export {
...
struct obd_device   *exp_obd;
...
}
```

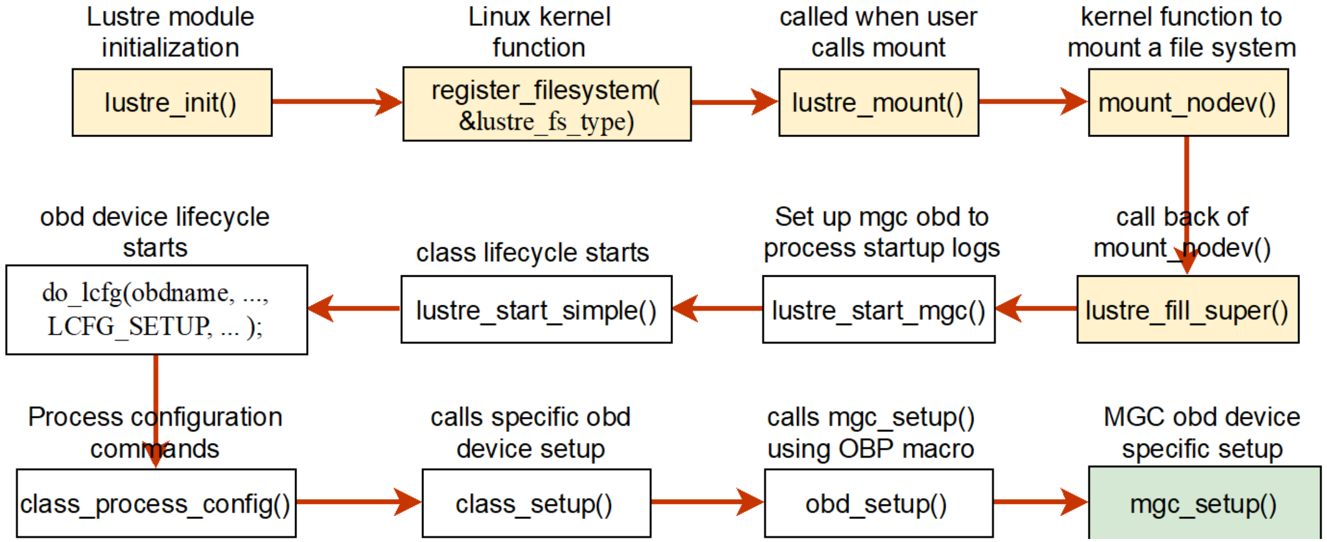
include/obd.h

```
struct obd_device {
...
struct obd_type                 *obd_type;
struct obd_uuid                  obd_uuid;
char obd_name[MAX_OBD_NAME];
...
}
```

Figure 2. Data structures involved in the communication between mgc and llite subsystems [1]

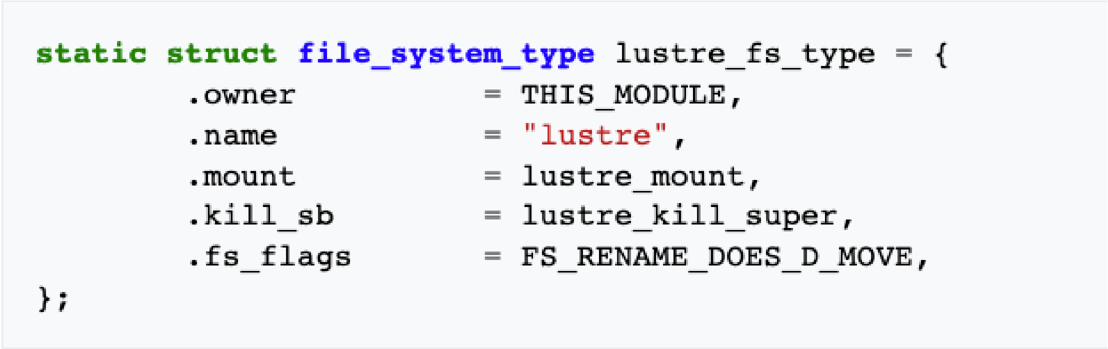
mgc_setup()

- The initial routine that gets executed to start and setup the MGC obd device
- Lustre module initialization begins from lustre_init()
- register_filesystem() registers Lustre among the list of file systems
- lustre_fill_super() is the entry point for mount call from client
- lustre_start_mgc() sets up MGC obd device to start process logs
- lustre_start_simple() invokes obdclass specific routines through do_lcfg()



Lustre module initialization
lustre_init()
Linux kernel function
register_filesystem(&lustre_fs_type)
called when user calls mount
lustre_mount()
kernel function to mount a file system
mount_nodev()
call back of mount_nodev()
lustre_fill_super()
Set up mgc obd to process startup logs
lustre_start_mgc()
class lifecycle starts
lustre_start_simple()
obd device lifecycle starts
do_lcfg(obdname, ..., LCFG_SETUP, ...);
Process configuration commands
class_process_config()
calls specific obd device setup
class_setup()
calls mgc_setup() using OBP macro
obd_setup()
MGC obd device specific setup
mgc_setup()

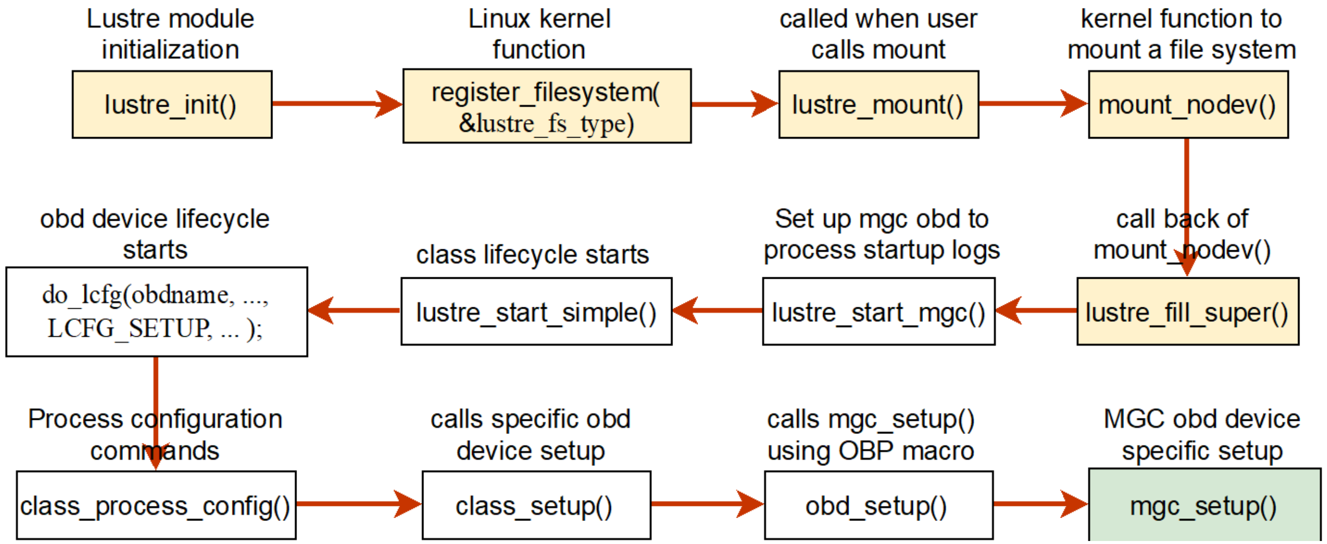
Figure 3. mgc_setup() call graph starting from Lustre file system mounting[1]



```
static struct file_system_type lustre_fs_type = {
        .owner          = THIS_MODULE,
        .name           = "lustre",
        .mount          = lustre_mount,
        .kill_sb        = lustre_kill_super,
        .fs_flags       = FS_RENAME_DOES_D_MOVE,
};
```

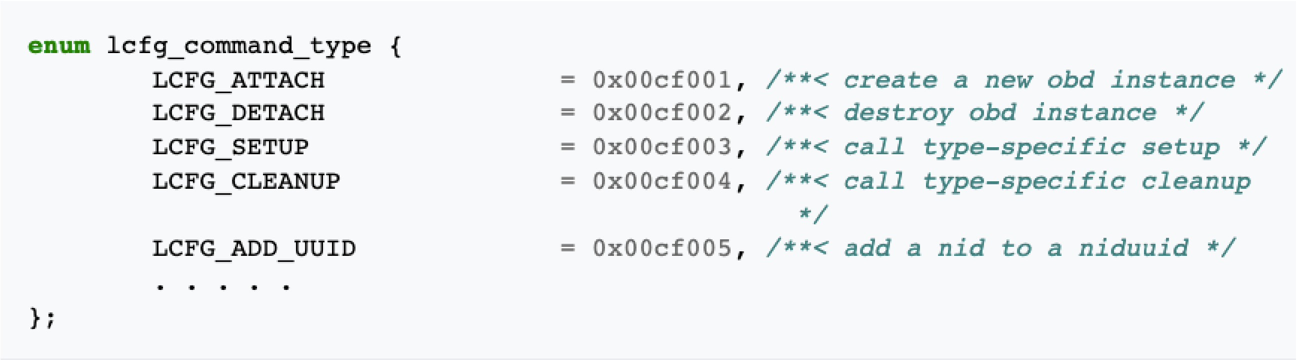
mgc_setup()(cont.)

- do_lcfg() takes obd device name and lcfg_command
- do_lcfg() takes LCFG_SETUP and invokes class_process_config()
- class_setup() creates hashes and self-export
- obd_setup() calls mgc_setup() through OBP macro
- mgc_setup() does the following,
  - Adds reference to PTL-RPC layer
  - Sets up RPC client using client_obd_setup()
  - mgc_llog_init() initializes Lustre logs
  - mgc_tunables_init() initializes the tunables
  - kthread_run starts mgc_requeue_thread



Lustre module initialization
lustre_init()
Linux kernel function
register_filesystem(&lustre_fs_type)
called when user calls mount
lustre_mount()
kernel function to mount a file system
mount_nodev()
call back of mount_nodev()
lustre_fill_super()
Set up mgc obd to process startup logs
lustre_start_mgc()
class lifecycle starts
lustre_start_simple()
obd device lifecycle starts
do_lcfg(obdname, ..., LCFG_SETUP, ...);
Process configuration commands
class_process_config()
calls specific obd device setup
class_setup()
calls mgc_setup() using OBP macro
obd_setup()
MGC obd device specific setup
mgc_setup()

Figure 3. mgc_setup() call graph starting from Lustre file system mounting [1]



```
enum lcfg_command_type {
        LCFG_ATTACH                  = 0x00cf001, /**< create a new obd instance */
        LCFG_DETACH                  = 0x00cf002, /**< destroy obd instance */
        LCFG_SETUP                   = 0x00cf003, /**< call type-specific setup */
        LCFG_CLEANUP                 = 0x00cf004, /**< call type-specific cleanup
                                                    */
        LCFG_ADD_UUID                = 0x00cf005, /**< add a nid to a niduuid */
        . . . . .
};
```

Lustre Log Handling

- Lustre makes use of logging for recovery and distributed transaction commits
- Logs associated with Lustre are called ‘`llogs`’
- Config logs, startup logs and change logs correspond to various `llogs`
- `llog_reader` can be used to read `llogs`
- MGS constructs a log for the target when Lustre target registers with MGS
- Lustre client log is created for client when its mounted
- When user mounts the the Lustre client, logs are downloaded on the client
- MGC reads and processes the logs and sends them to clients and servers

Log Processing in MGC

- lustre_fill_super() makes a call to ll_fill_super()
- Initializes a config log instance specific to super block
- lustre_process_log() gets a config log from MGS and starts processing it
- Continues to process new statements appended to logs
- Resets lustre_cfg_bufs and calls obd_process_config()
- Invokes mgc_process_config() using OBP macro
- The lcfg_command passed is LCFG_LOG_START
- config_log_add() adds the log to the list of active logs watched for updates by MGC

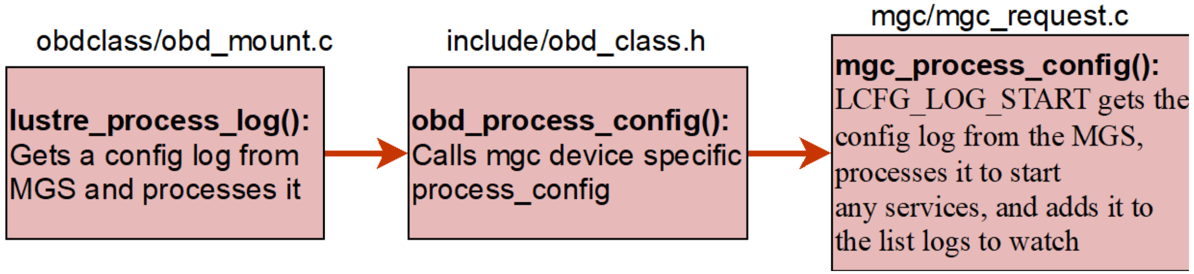

obdclass/obd_mount.c
lustre_process_log(): Gets a config log from MGS and processes it
include/obd_class.h
obd_process_config(): Calls mgc device specific process_config
mgc/mgc_request.c
mgc_process_config(): LCFG_LOG_START gets the config log from the MGS, processes it to start any services, and adds it to the list logs to watch

Figure 4. mgc_process_config() call graph [1]

mgc_precleanup() and mgc_cleanup()

- class_cleanup() starts the shutdown of an obd device
- mgc_precleanup() makes sure that all exports are destroyed
- Decrements mgc_count which keeps count of number of threads
- obd_cleanup_client_import() destroys client-side import interface
- mgc_cleanup() invokes mgc_llog_fini() which cleans up Lustre logs with the MGC
- Log cleaning is accomplished by llog_cleanup()

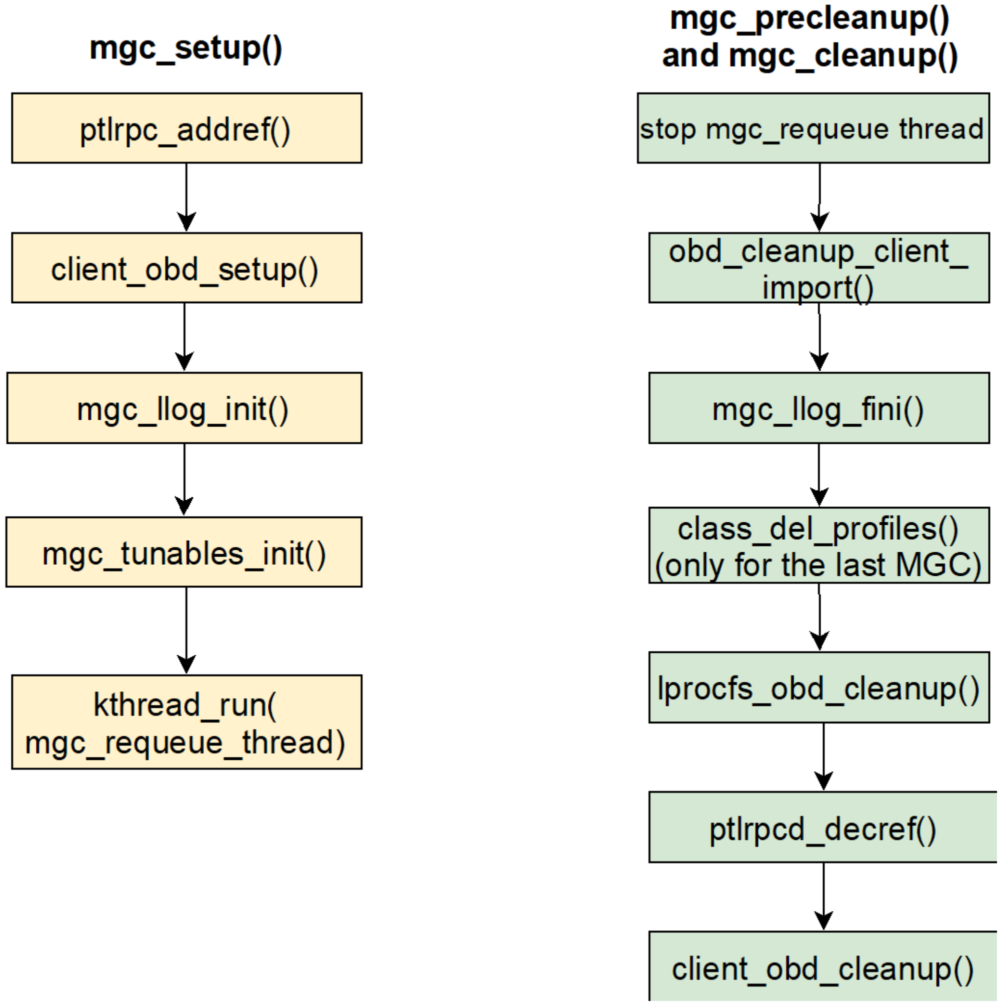


Figure 4. mgc_setup() vs. mgc_cleanup() [1]

mgc_precleanup() and mgc_cleanup() (cont.)

- mgc_cleanup() deletes profiles for the last MGC obd device using class_del_profiles()
- Lustre profiles help to identify intended recipients of the data
- lprocsfs_obd_cleanup() removes sysfs and debugfs entries
- Decrements reference to PTL-RPC layer and calls client_obd_cleanup()
- Makes the obd namespace points to NULL and destroys client-side import interface
- Frees up the obd device using OBD_FREE macro
- After the obd_precleanup(), uuid-export and nid-export hashtables are freed up and destroyed

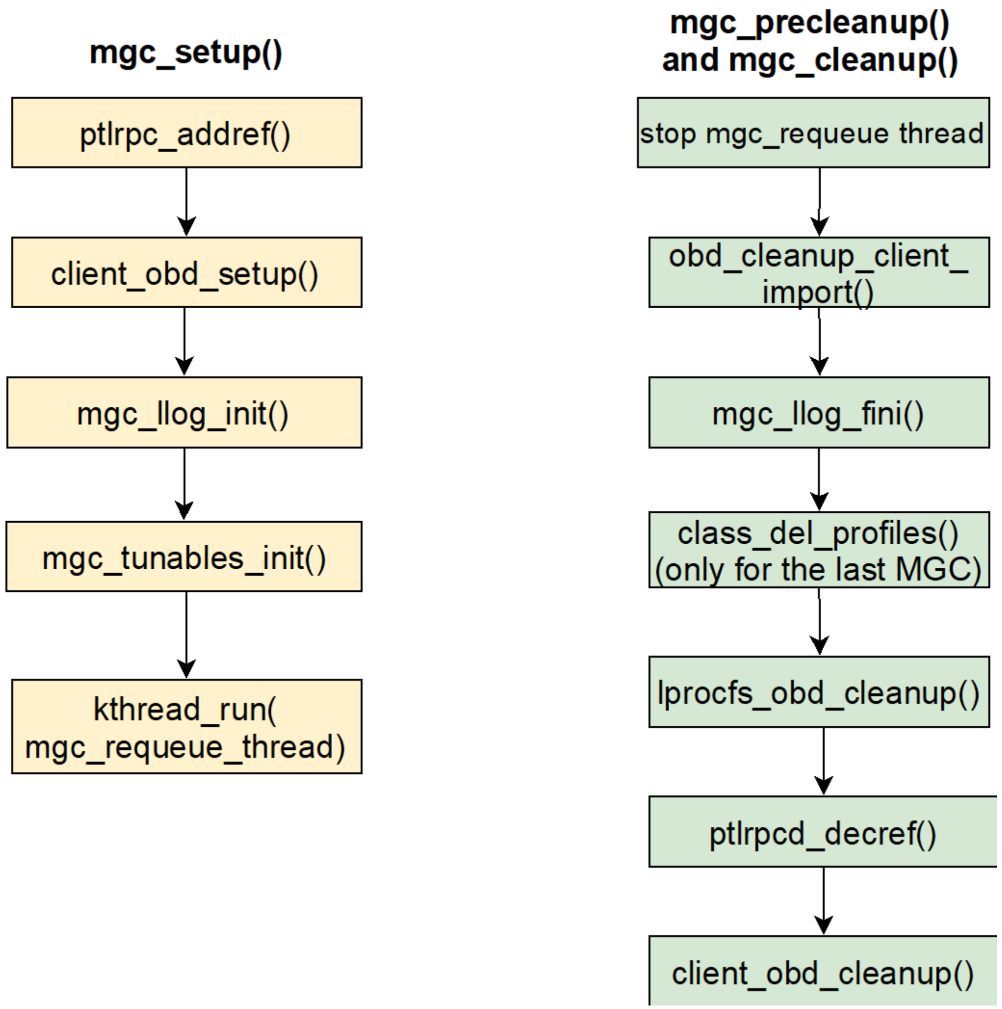
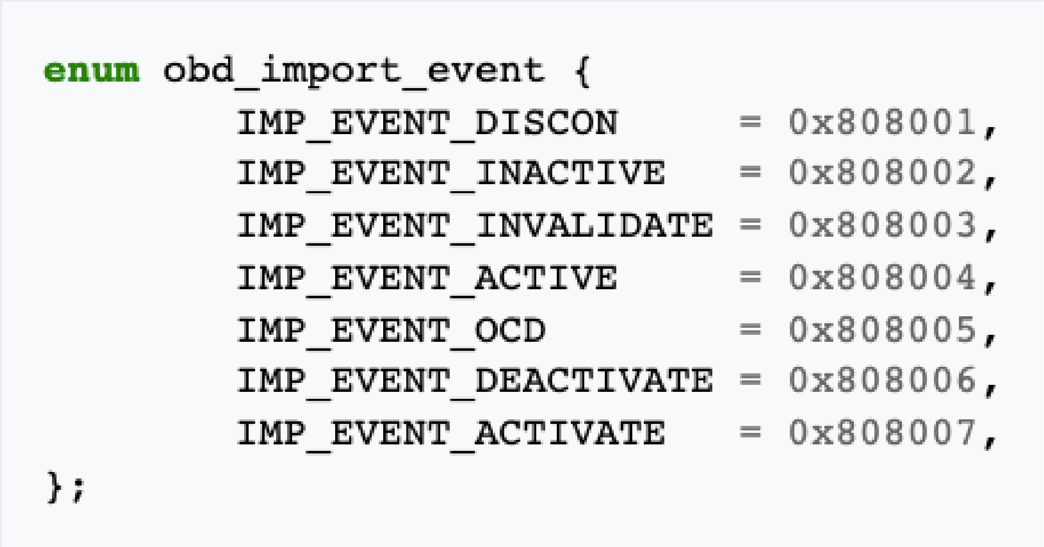


Figure 4. mgc_setup() vs. mgc_cleanup() [1]

mgc_import_event()

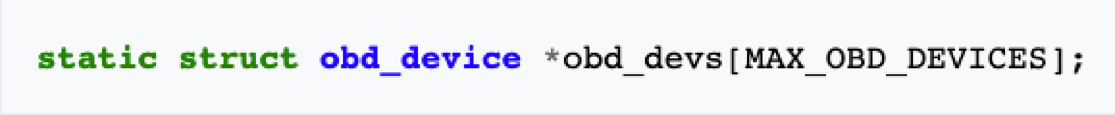
- mgc_import_event() handles events reported at the MGC import interface
- Type of import events identified by MGC are listed in obd_import_event
- Client-side imports are used by clients to communicate with exports on the server
- For e.g., in communication from MDS to MGS, MDS will be using client import to communicate with MGS server-side export



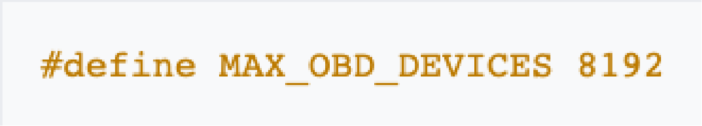
```
enum obd_import_event {
        IMP_EVENT_DISCON     = 0x808001,
        IMP_EVENT_INACTIVE   = 0x808002,
        IMP_EVENT_INVALIDATE = 0x808003,
        IMP_EVENT_ACTIVE     = 0x808004,
        IMP_EVENT_OCD        = 0x808005,
        IMP_EVENT_DEACTIVATE = 0x808006,
        IMP_EVENT_ACTIVATE   = 0x808007,
};
```

Introduction to Obdclass

- Obdclass allows to apply generic operations without knowing the specific details of obd devices
- MGC, MDC, OSC, LOV and LMV are examples of obd devices in Lustre that makes use of obdclass
- Obd devices can be connected in different ways to form client-server pairs for data exchange in Lustre
- Obd devices in Lustre are stored in obd_devs array and is limited by `MAX_OBD_DEVICES`
- obd_devs array is indexed using obd_`minor` number
- An obd device can be identified using its minor number, name or uuid



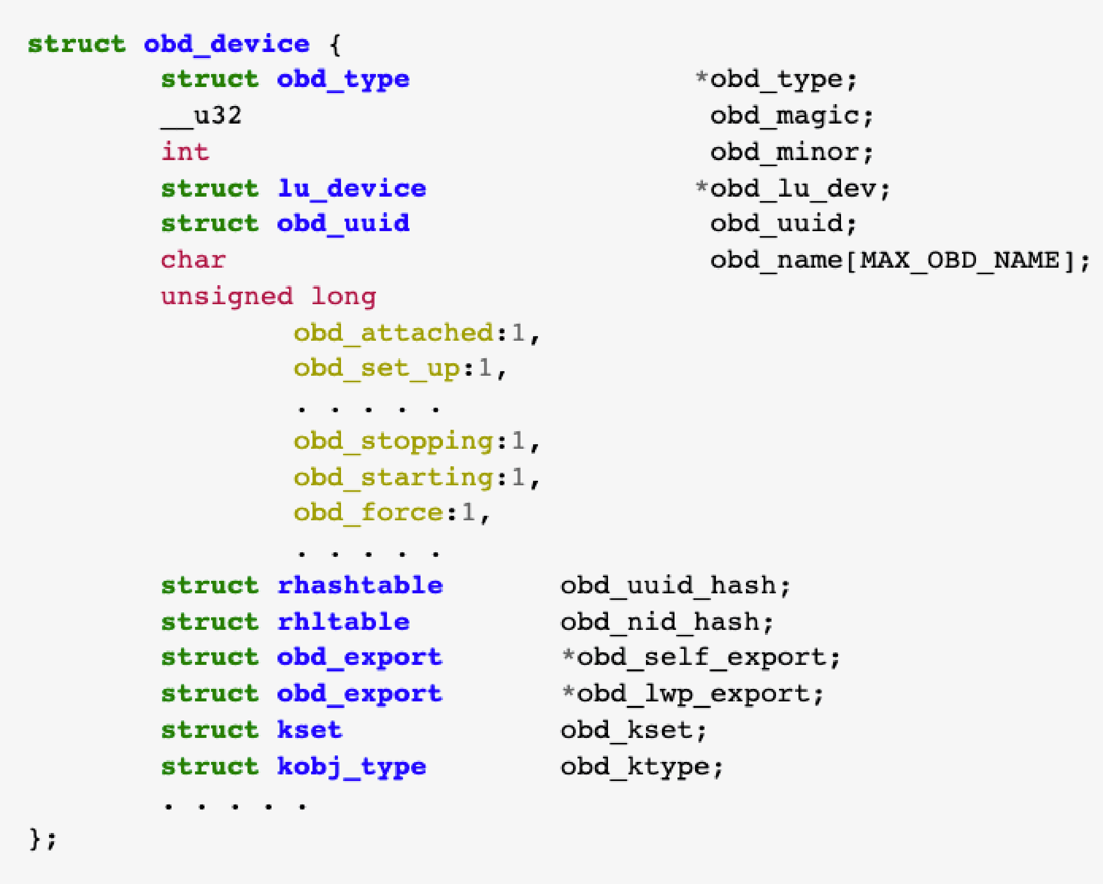
```
static struct obd_device *obd_devs[MAX_OBD_DEVICES];
```



```
#define MAX_OBD_DEVICES 8192
```

obd_device structure

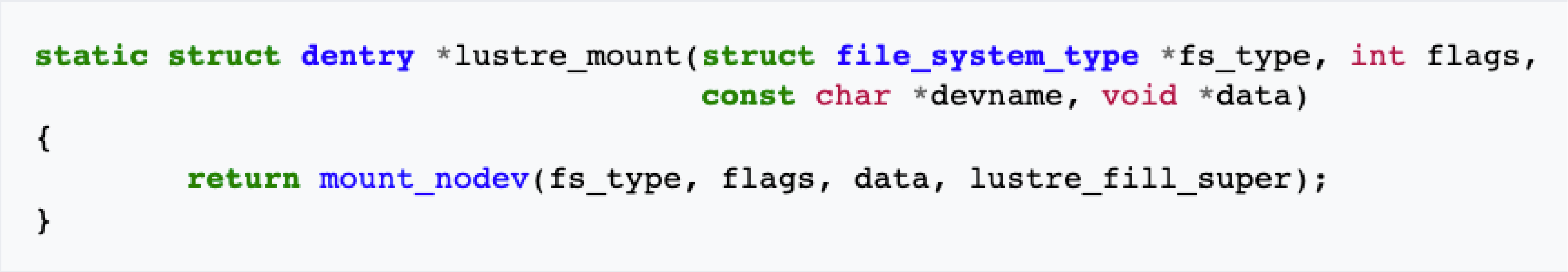
- obd_type defines the type of the obd device – metadata, bulk or both
- obd_magic used to identify data corruption with an obd device
- obd_minor is the index of obd_devs array
- lu_device indicates obd device is a real device such as ldiskfs or zfs
- Includes various flags to indicate the current status of obd device
  - obd_attached, obd_set_up, obd_stopping, obd_starting and so on
- uuid-export and nid-export hash tables for obd device
- Linked lists pointing to obd_nid_stats, obd_exports and obd_unlinked_exports



```
struct obd_device {
        struct obd_type                *obd_type;
        __u32                           obd_magic;
        int                             obd_minor;
        struct lu_device               *obd_lu_dev;
        struct obd_uuid                 obd_uuid;
        char                            obd_name[MAX_OBD_NAME];
        unsigned long
                obd_attached:1,
                obd_set_up:1,
                . . . . .
                obd_stopping:1,
                obd_starting:1,
                obd_force:1,
                . . . . .
        struct rhashtable       obd_uuid_hash;
        struct rhltable         obd_nid_hash;
        struct obd_export       *obd_self_export;
        struct obd_export       *obd_lwp_export;
        struct kset             obd_kset;
        struct kobj_type        obd_ktype;
        . . . . .
};
```

MGC Life Cycle

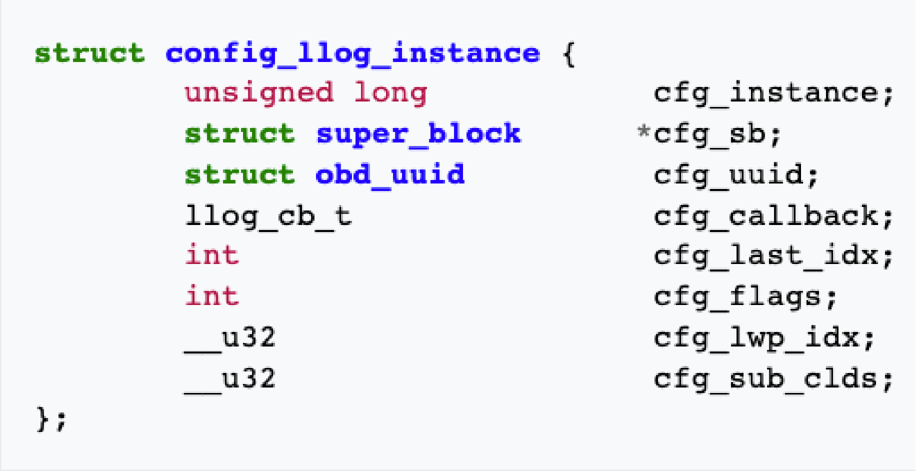
- MGC is the first obd device setup and started by Lustre in the life cycle
- Generic file system mount function `vfs_mount()` is invoked by mount system call from user
  - Handles the generic portion of mounting the file system
- Invokes specific mount function, `lustre_mount()`
- `lustre_mount()` invokes kernel function `mount_nodev()` which invokes `lustre_fill_super()` as its callback function



```
static struct dentry *lustre_mount(struct file_system_type *fs_type, int flags,
                                   const char *devname, void *data)
{
        return mount_nodev(fs_type, flags, data, lustre_fill_super);
}
```

MGC Life Cycle

- `lustre_fill_super()` is the entry point for mount call into Lustre
- Initializes Lustre superblock, which is used by MGC to write a local copy of the config log
- `ll_fill_super()` initializes a config log instance specific for the superblock
- `cfg_instance` field is unique to the superblock, obtained using `ll_get_cfg_instance()`
- Also has a uuid and a callback handler defined by the function `class_config_llog_handler()`
- File system name field of `ll_sb_info` is populated by using `get_profile_name()`
- `get_profile_name()` obtains a profile name corresponding to the mount command issued from the user from the `lustre_mount_data` structure



```
struct config_llog_instance {
        unsigned long           cfg_instance;
        struct super_block     *cfg_sb;
        struct obd_uuid         cfg_uuid;
        llog_cb_t               cfg_callback;
        int                     cfg_last_idx;
        int                     cfg_flags;
        __u32                   cfg_lwp_idx;
        __u32                   cfg_sub_clds;
};
```

MGC Life Cycle

- `ll_fill_super()`invokes `lustre_process_log()` which gets config logs from MGS
- Will continue to process new statements appended to the logs
- Three parameters passed to this function are superblock, log name and config log instance
  - config log instance is used by MGC to write local copy of the config log
  - logname is name of llog replicated from MGS
- `obd_process_config()` uses OBP macro to call MGC specific `mgc_process_config()`
- Logs are added to the list of logs to watch

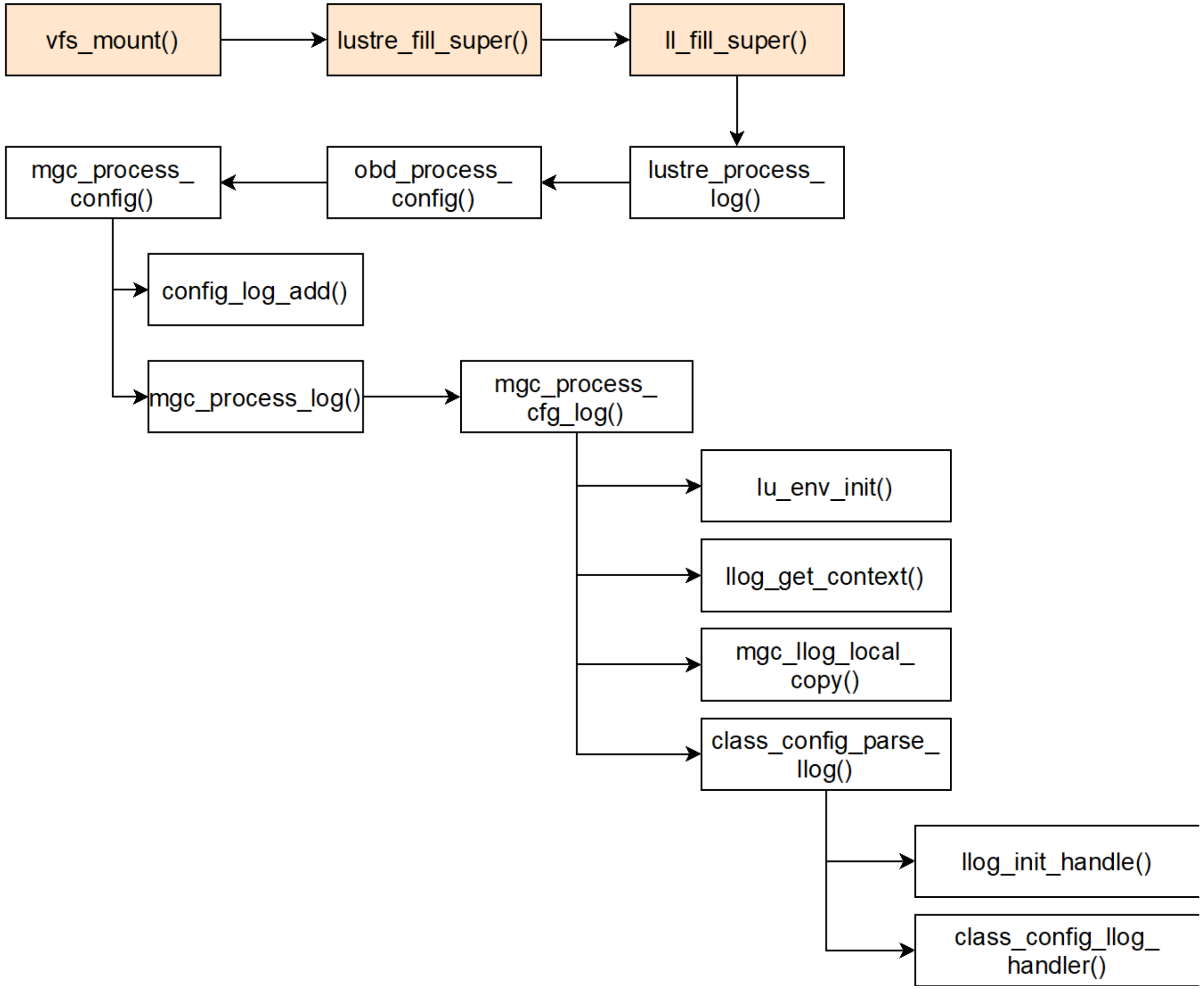


Figure 5. Obd device life cycle workflow for MGC [1]

MGC Life Cycle

- mgc_process_config() invokes the following subfunctions
- config_log_add() categorizes the data in config log based on:
  - ptl-rpc layer, configuration parameters, nodemaps and barriers
- Log data related to each of these is copied to memory using config_log_find_or_add()
- mgc_process_log() gets config log from MGS
- mgc_process_cfg_log() reads the log and creates a local copy
- Initializes an environment and context using lu_env_init() and llog_get_context()
- mgc_llog_local_copy() is used to create a local copy of the log

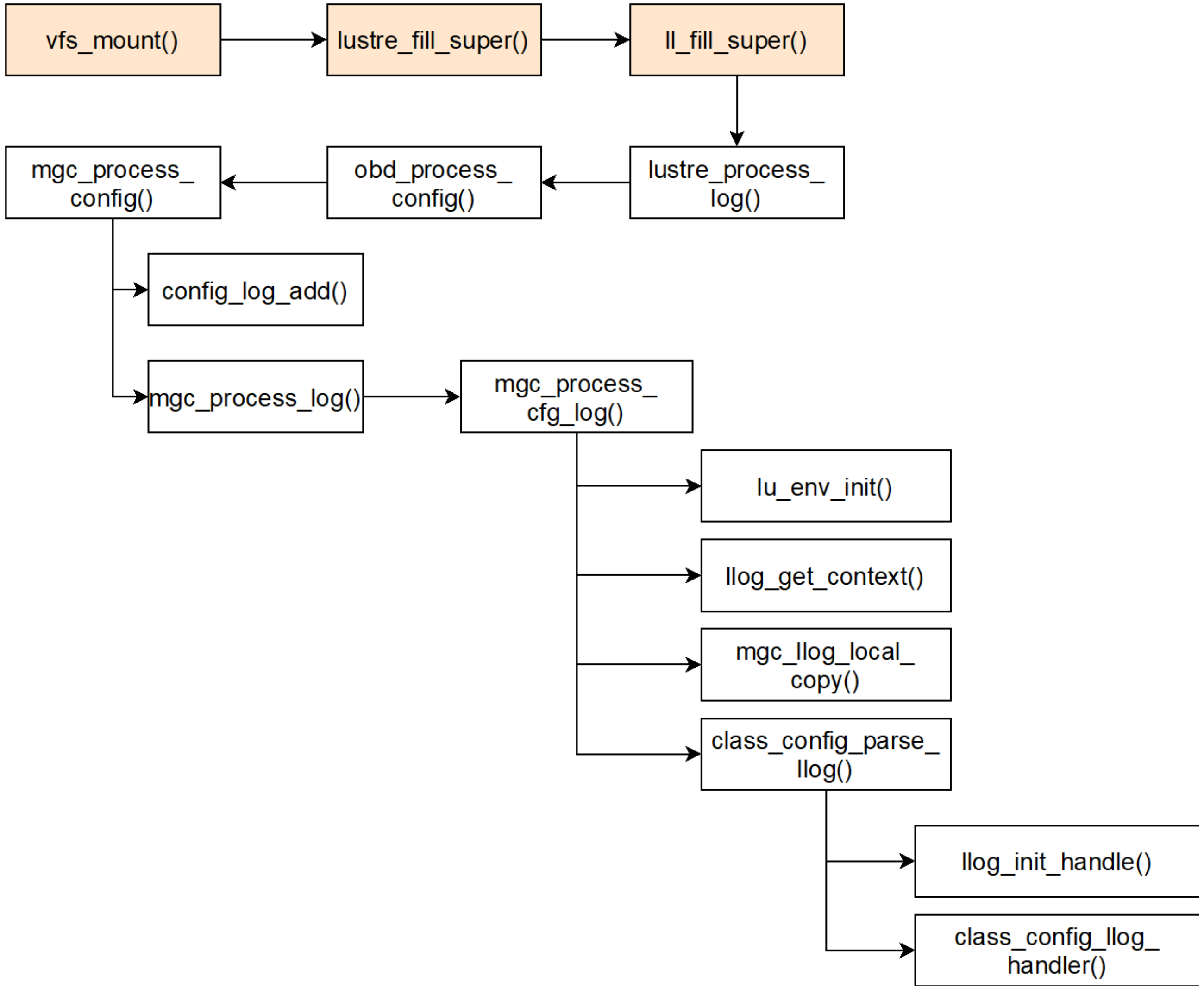


Figure 5. Obd device life cycle workflow for MGC [1]

MGC Life Cycle

- Real time changes are parsed using the function class_config_parse_llog()
- First log that is being parsed is start_log
- class_config_parse_llog() acquires lock on the log using llog_init_handle()
- Uses an index and a callback function (class_config_llog_handler()) to process logs
- Callback handler also initializes lustre_cfg_bufs to temporarily store log data
- The following actions take place afterwards,
  - translates log names to obd device names
  - appends uuid with obd device name for each Lustre client mount
  - attaches the obd device

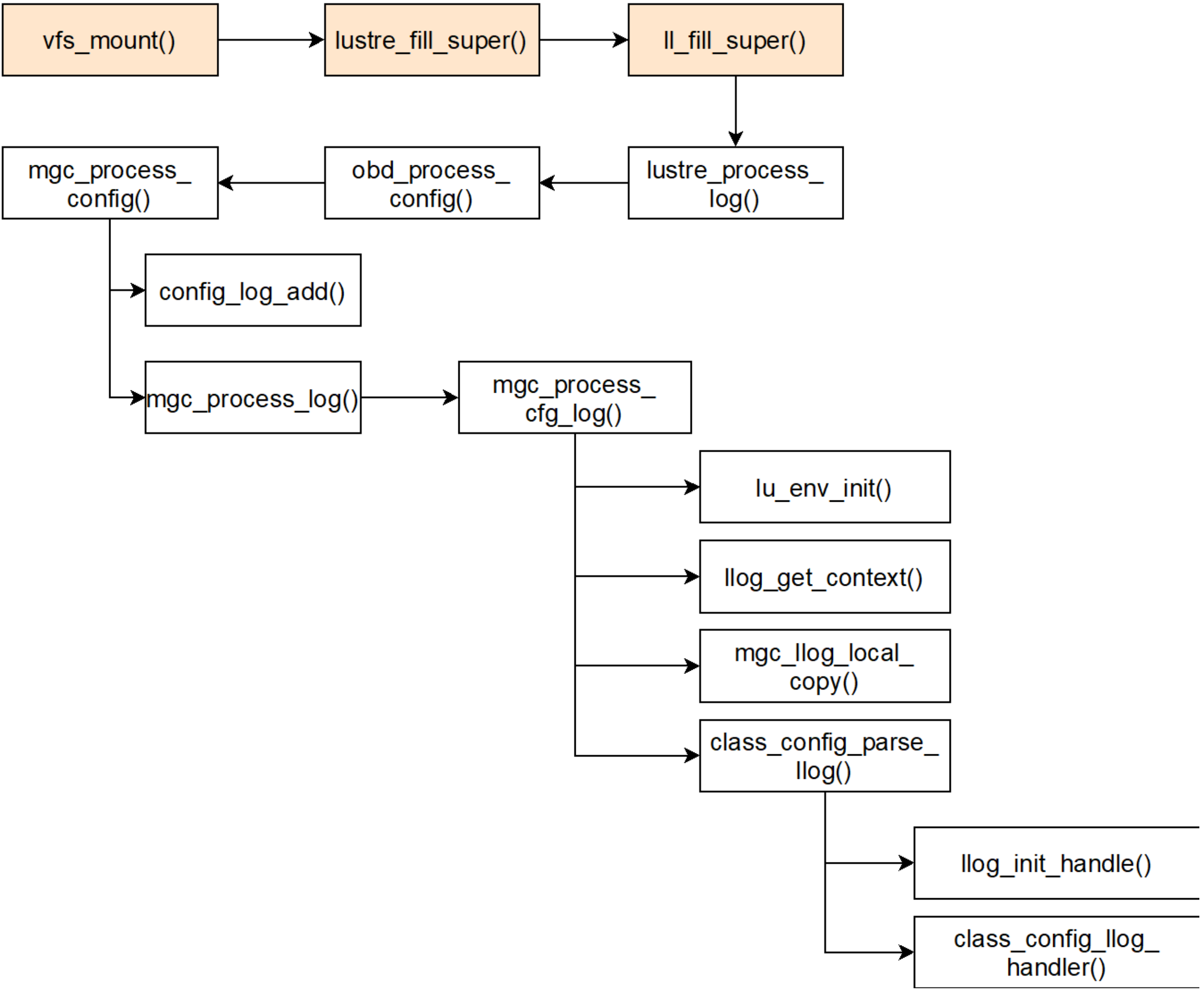


Figure 5. Obd device life cycle workflow for MGC [1]

MGC Life Cycle

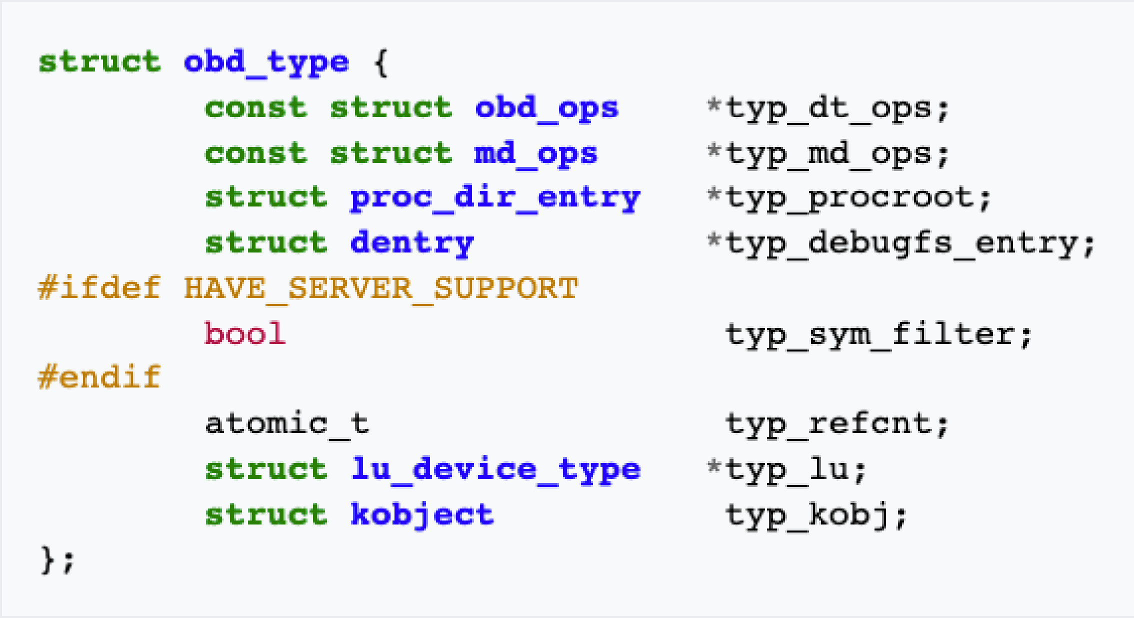
- Each obd device then sets up a key to communicate with other devices through secure ptl-rpc layer
- Rules for creating this key are stored in the config log
- Obd device then creates a connection for communication
- Start log contains all state information for all configuration devices
- The lustre configuration buffer (`lustre_cfg_bufs`) stores this information temporarily
- Obd device then use this buffer to consume log data
- After creating a connection, the handler extracts information (uuid, nid etc.) required to form Lustre `config_logs`

MGC Life Cycle

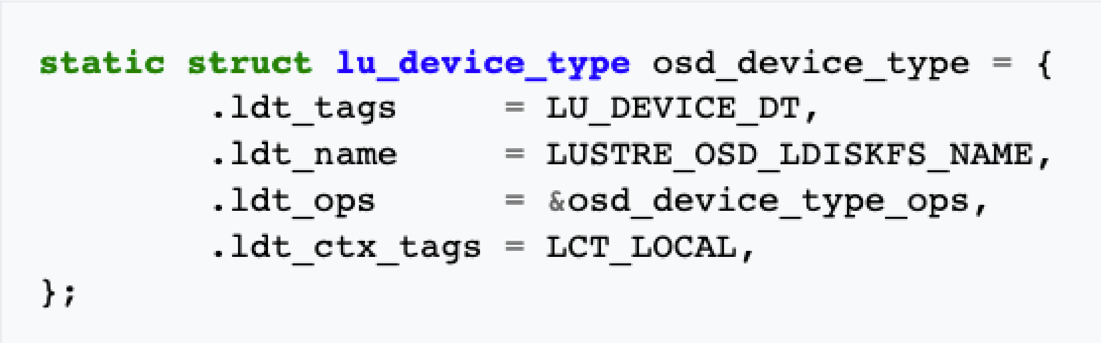
- `llog_rec_hdr` decides what type of information should be parsed from the logs
- For instance,
  - `OBD_CFG_REC` indicates the handler to scan obd device configuration information
  - `CHANGELOG_REC` asks to parse for changelog records
- Using nid and uuid information about the obd device the handler now invokes `class_process_config()` routine
- This function repeats the cycle of obd device creation for other obd devices
- Notice that the only obd device exists in Lustre at this point in the life cycle is MGC
- `class_process_config()` function calls the generic obd class functions such as `class_attach()`, and `class_setup()` depending upon the `lcfg_command` it receives for a specific obd device

Obd Device Life Cycle – `class_attach()`

- First method in the lifecycle of an obd device – `class_attach()`
- Registers and adds the obd device to the list of obd devices
- The attach function checks if obd type being passed is valid
- `obd_ops` and `md_ops` determine obd device performs what type of operation
- `lu_device_type` is applicable only for real block devices
- Differentiates metadata and bulk data devices using `LU_DEVICE_MD` and `LU_DEVICE_DT`



```
struct obd_type {
        const struct obd_ops    *typ_dt_ops;
        const struct md_ops     *typ_md_ops;
        struct proc_dir_entry   *typ_procroot;
        struct dentry           *typ_debugfs_entry;
#ifdef HAVE_SERVER_SUPPORT
        bool                     typ_sym_filter;
#endif
        atomic_t                 typ_refcnt;
        struct lu_device_type   *typ_lu;
        struct kobject           typ_kobj;
};
```



```
static struct lu_device_type osd_device_type = {
        .ldt_tags     = LU_DEVICE_DT,
        .ldt_name     = LUSTRE_OSD_LDISKFS_NAME,
        .ldt_ops      = &osd_device_type_ops,
        .ldt_ctx_tags = LCT_LOCAL,
};
```

Obd Device Life Cycle – `class_attach()`

- `class_newdev()` creates and allocates a new obd device
- `class_get_type()` registers and loads the device
- `class_new_export_self()` creates a new export and adds it to the hashtable of exports
- Self-export is created only for client obd device
- Last part of `class_attach()` is registering/listing obd device
- `class_register_device()` lists the device in obd_devs array
- Assigns a minor number to the obd device

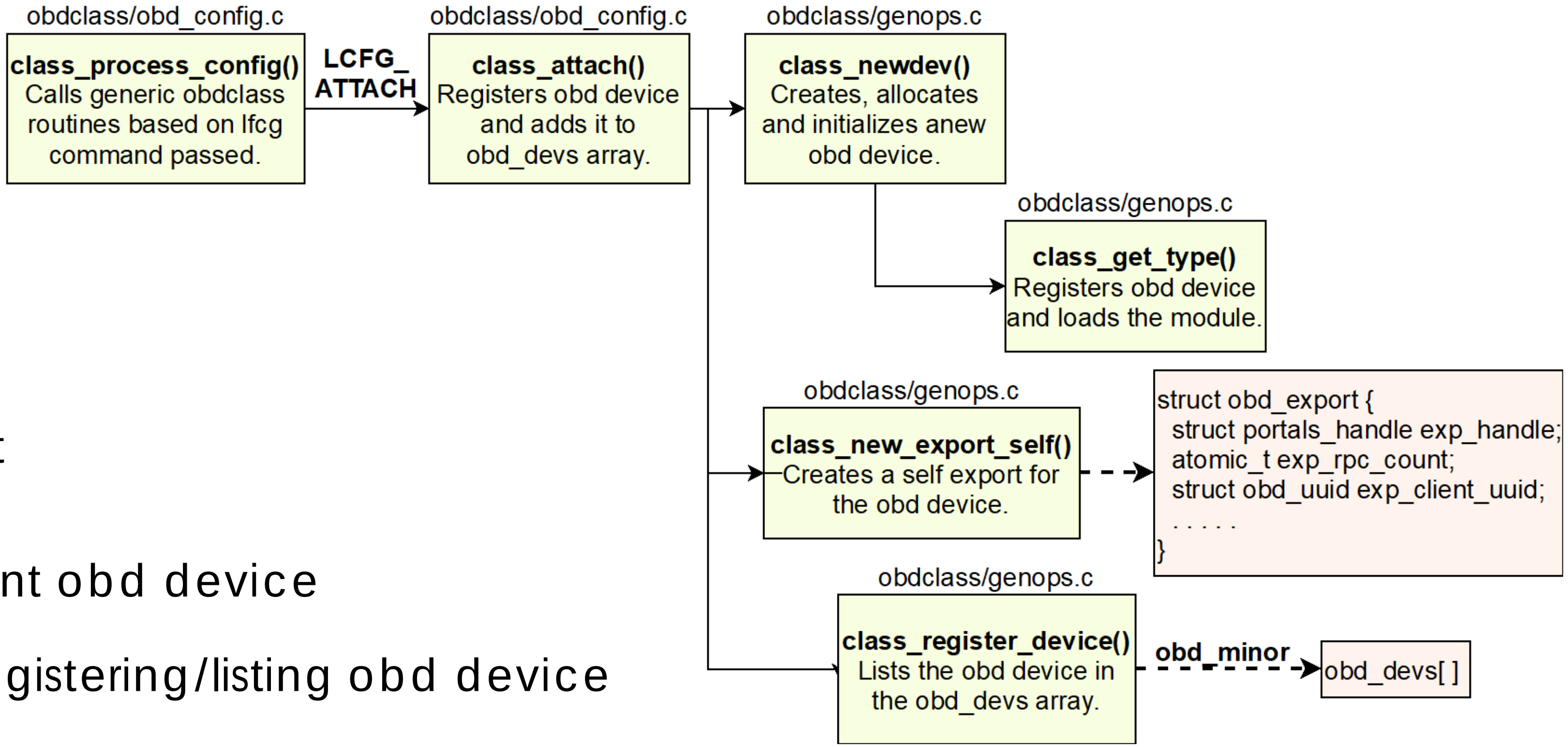
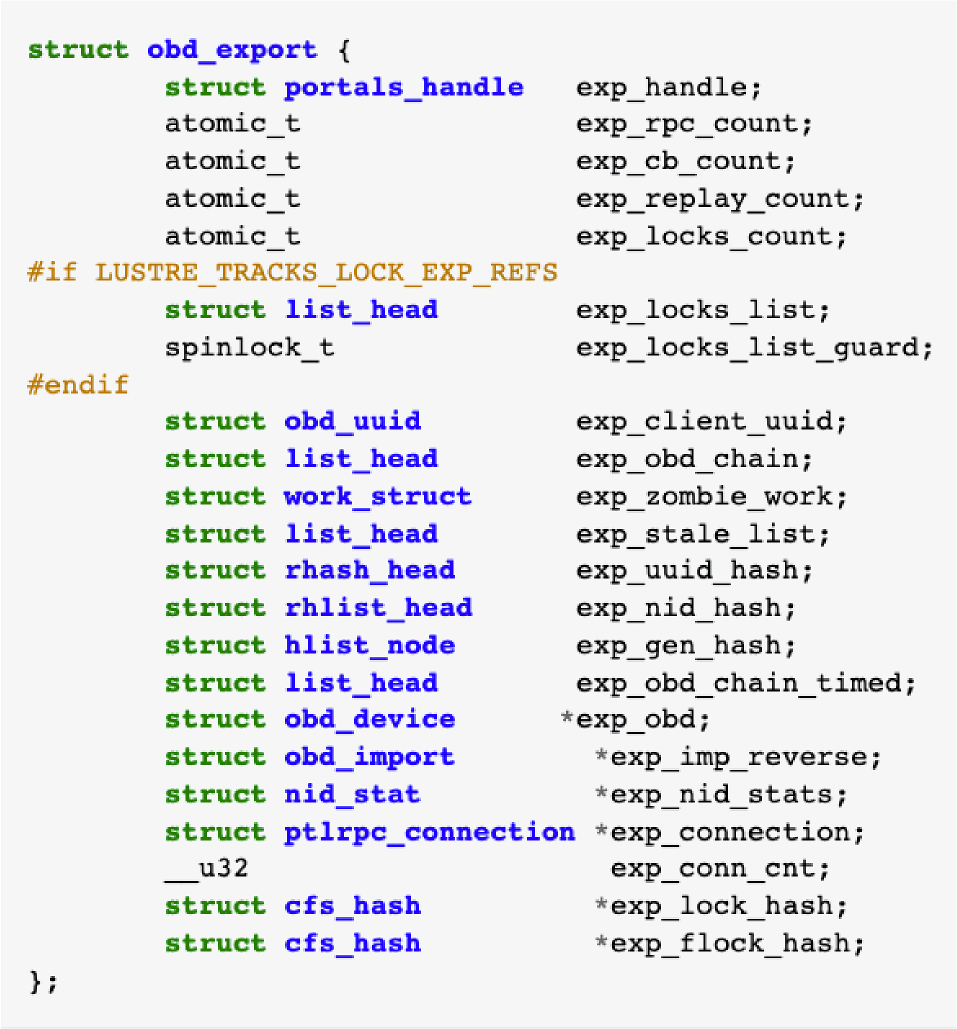

obdclass/obd_config.c
class_process_config()
Calls generic obdclass routines based on lfcg command passed.
LCFG_ ATTACH
obdclass/obd_config.c
class_attach()
Registers obd device and adds it to obd_devs array.
obdclass/genops.c
class_newdev()
Creates, allocates and initializes anew obd device.
obdclass/genops.c
class_get_type()
Registers obd device and loads the module.
obdclass/genops.c
class_new_export_self()
Creates a self export for the obd device.
struct obd_export {
struct portals_handle exp_handle;
atomic_t exp_rpc_count;
struct obd_uuid exp_client_uuid;
. . . . .
}
obdclass/genops.c
class_register_device()
Lists the obd device in the obd_devs array.
obd_minor
obd_devs[ ]

Figure 6. Workflow of class_attach() function in obd device lifecycle [1]

obd_export Structure

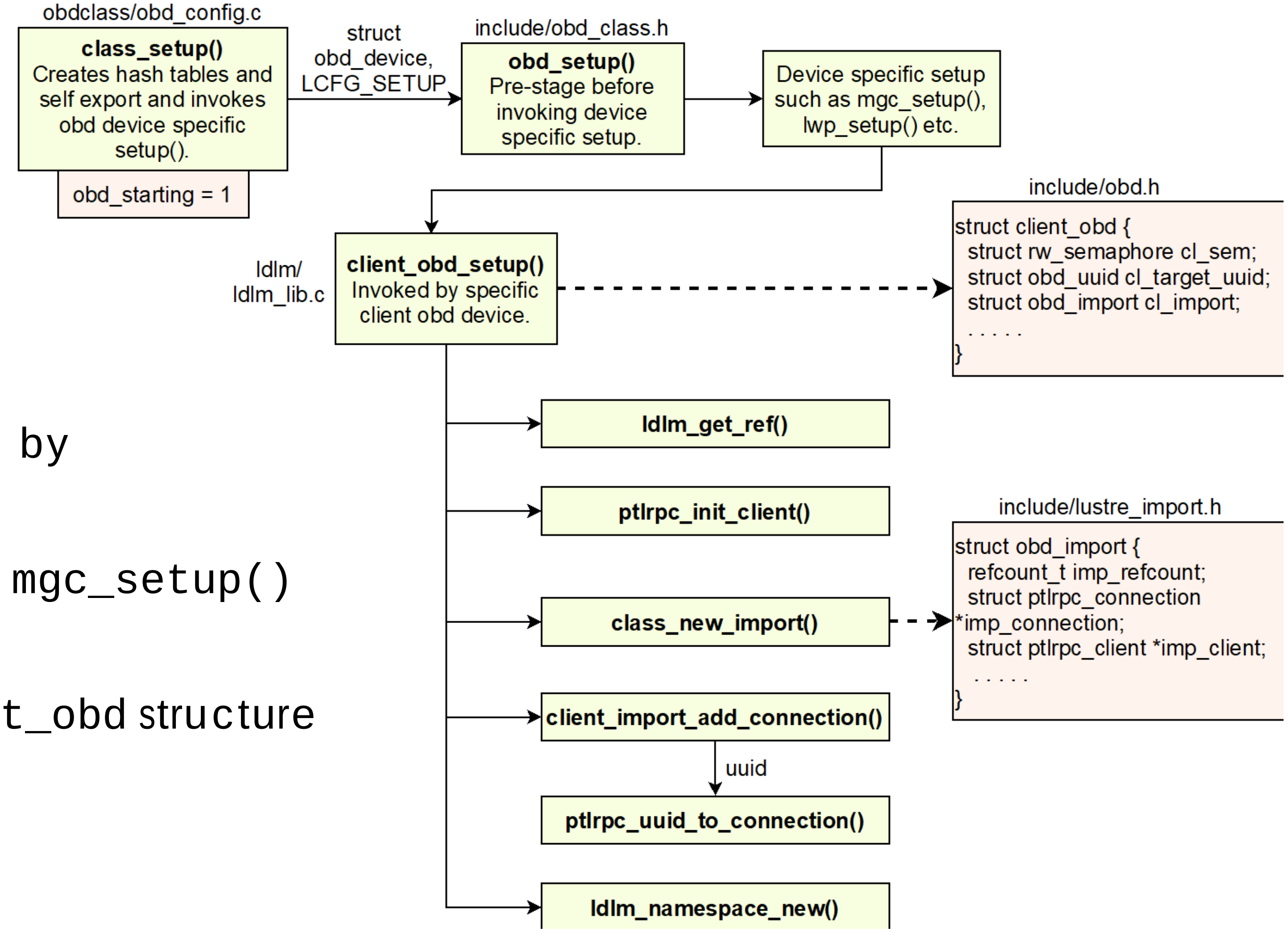
- obd_export represents a target side export connection for an obd device
- For every connected client, there will be an export structure on the server attached to the same obd device
- exp_handle is used identify which export the clients are talking to
- exp_rpc_count is the number of RPC references
- exp_cb_count counts commit callback references
- exp_locks_list maintains a linked list of all the locks
- exp_client_uuid is the UUID of client connected to this export
- exp_obd_chain links all the exports on an obd device



```
struct obd_export {
        struct portals_handle   exp_handle;
        atomic_t                exp_rpc_count;
        atomic_t                exp_cb_count;
        atomic_t                exp_replay_count;
        atomic_t                exp_locks_count;
#if LUSTRE_TRACKS_LOCK_EXP_REFS
        struct list_head        exp_locks_list;
        spinlock_t              exp_locks_list_guard;
#endif
        struct obd_uuid         exp_client_uuid;
        struct list_head        exp_obd_chain;
        struct work_struct      exp_zombie_work;
        struct list_head        exp_stale_list;
        struct rhash_head       exp_uuid_hash;
        struct rhlist_head      exp_nid_hash;
        struct hlist_node       exp_gen_hash;
        struct list_head        exp_obd_chain_timed;
        struct obd_device      *exp_obd;
        struct obd_import       *exp_imp_reverse;
        struct nid_stat         *exp_nid_stats;
        struct ptlrpc_connection *exp_connection;
        __u32                    exp_conn_cnt;
        struct cfs_hash          *exp_lock_hash;
        struct cfs_hash          *exp_flock_hash;
};
```

Obd Device Life Cycle – class_setup()

- class_setup() creates hashtables and self-export
- Obtains obd device from obd_devs array using obd_minor
- Sets obd_starting flag to indicate that set up of this device has started
- Device specific obd_setup() is invoked by class_setup()
- Invokes device specific routines such as mgc_setup() and lwp_setup()
- client_obd_setup() populates client_obd structure

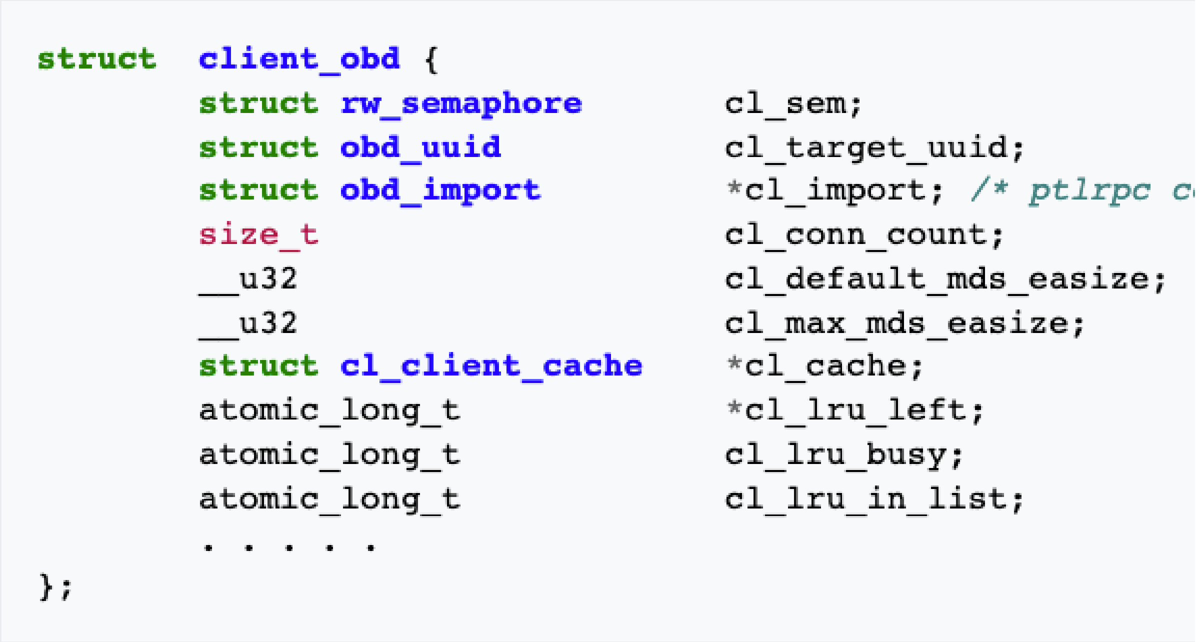


obdclass/obd_config.c
class_setup()
Creates hash tables and self export and invokes obd device specific setup().
obd_starting = 1
struct obd_device, LCFG_SETUP
include/obd_class.h
obd_setup()
Pre-stage before invoking device specific setup.
Device specific setup such as mgc_setup(), lwp_setup() etc.
ldlm/ ldlm_lib.c
client_obd_setup()
Invoked by specific client obd device.
include/obd.h
struct client_obd {
struct rw_semaphore cl_sem;
struct obd_uuid cl_target_uuid;
struct obd_import cl_import;
. . . . .
}
ldlm_get_ref()
ptlrpc_init_client()
class_new_import()
include/lustre_import.h
struct obd_import {
refcount_t imp_refcount;
struct ptlrpc_connection *imp_connection;
struct ptlrpc_client *imp_client;
. . . . .
}
client_import_add_connection()
uuid
ptlrpc_uuid_to_connection()
ldlm_namespace_new()

Figure 7. Workflow of class_setup() function in obd device lifecycle [1]

Obd Device Life Cycle – class_setup()

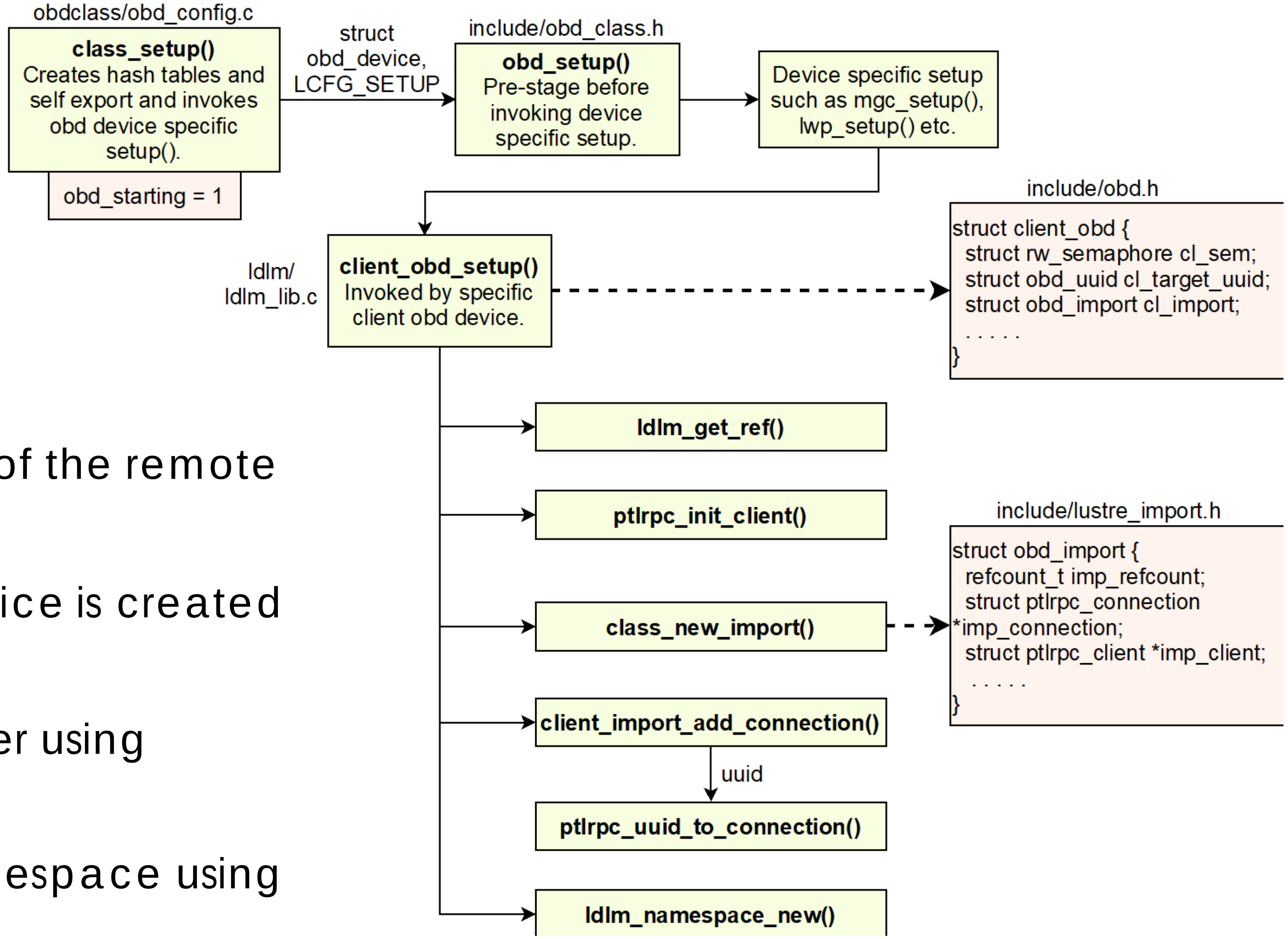
- client_obd structure is used for page cache and extended attributes management
- It comprises of fields pointing to,
  - uuid and import interfaces
  - counter to keep track of client connections
  - maximum and default extended attribute sizes
  - cl_cache: LRU cache for caching OSC pages
  - cl_lru_left: available LRU slots per OSC cache
  - cl_lru_busy: number of busy LRU pages
  - cl_lru_in_list: number of LRU pages in the cache for client_obd



```
struct  client_obd {
        struct rw_semaphore       cl_sem;
        struct obd_uuid           cl_target_uuid;
        struct obd_import         *cl_import; /* ptlrpc c
        size_t                    cl_conn_count;
        __u32                     cl_default_mds_easize;
        __u32                     cl_max_mds_easize;
        struct cl_client_cache    *cl_cache;
        atomic_long_t             *cl_lru_left;
        atomic_long_t             cl_lru_busy;
        atomic_long_t             cl_lru_in_list;
        . . . . .
};
```

Obd Device Life Cycle – `class_setup()`

- `client_obd_setup()` obtains LDLM lock to setup LDLM layer references
- Sets up ptl-rpc request and reply portals using `ptlrpc_init_client()`
- `client_obd` defines a pointer to the `obd_import` structure
- `obd_import` represents client-side view of the remote target
- New import connection for the obd device is created using `class_new_import()`
- Adds an initial connection to ptl-rpc layer using `client_import_add_connection()`
- `client_obd_setup()` creates ldlm namespace using `ldlm_namespace_new()`



obdclass/obd_config.c
class_setup()
Creates hash tables and self export and invokes obd device specific setup().
obd_starting = 1
struct obd_device, LCFG_SETUP
include/obd_class.h
obd_setup()
Pre-stage before invoking device specific setup.
Device specific setup such as mgc_setup(), lwp_setup() etc.
ldlm/ ldlm_lib.c
client_obd_setup()
Invoked by specific client obd device.
include/obd.h
ldlm_get_ref()
ptlrpc_init_client()
class_new_import()
include/lustre_import.h
client_import_add_connection()
uuid
ptlrpc_uuid_to_connection()
ldlm_namespace_new()

```
struct client_obd {
  struct rw_semaphore cl_sem;
  struct obd_uuid cl_target_uuid;
  struct obd_import cl_import;
  . . . . .
}
```

```
struct obd_import {
  refcount_t imp_refcount;
  struct ptlrpc_connection
*imp_connection;
  struct ptlrpc_client *imp_client;
   . . . . .
}
```

Figure 7. Workflow of class_setup() function in obd device lifecycle [1]

Obd Device Life Cycle – `class_precleanup()` and `class_cleanup()`

- Lustre unmount process begins from `ll_umount_begin()` defined as part of `ll_super_operations`
- `ll_umount_begin()` accepts a `super_block`
- Metadata and data exports are extracted using `class_exp2obd()`
- `obd_force` flag from `obd_device` structure is set to indicate the cleanup process
- Periodically checks and waits to finish until there are no outstanding requests from vfs layer

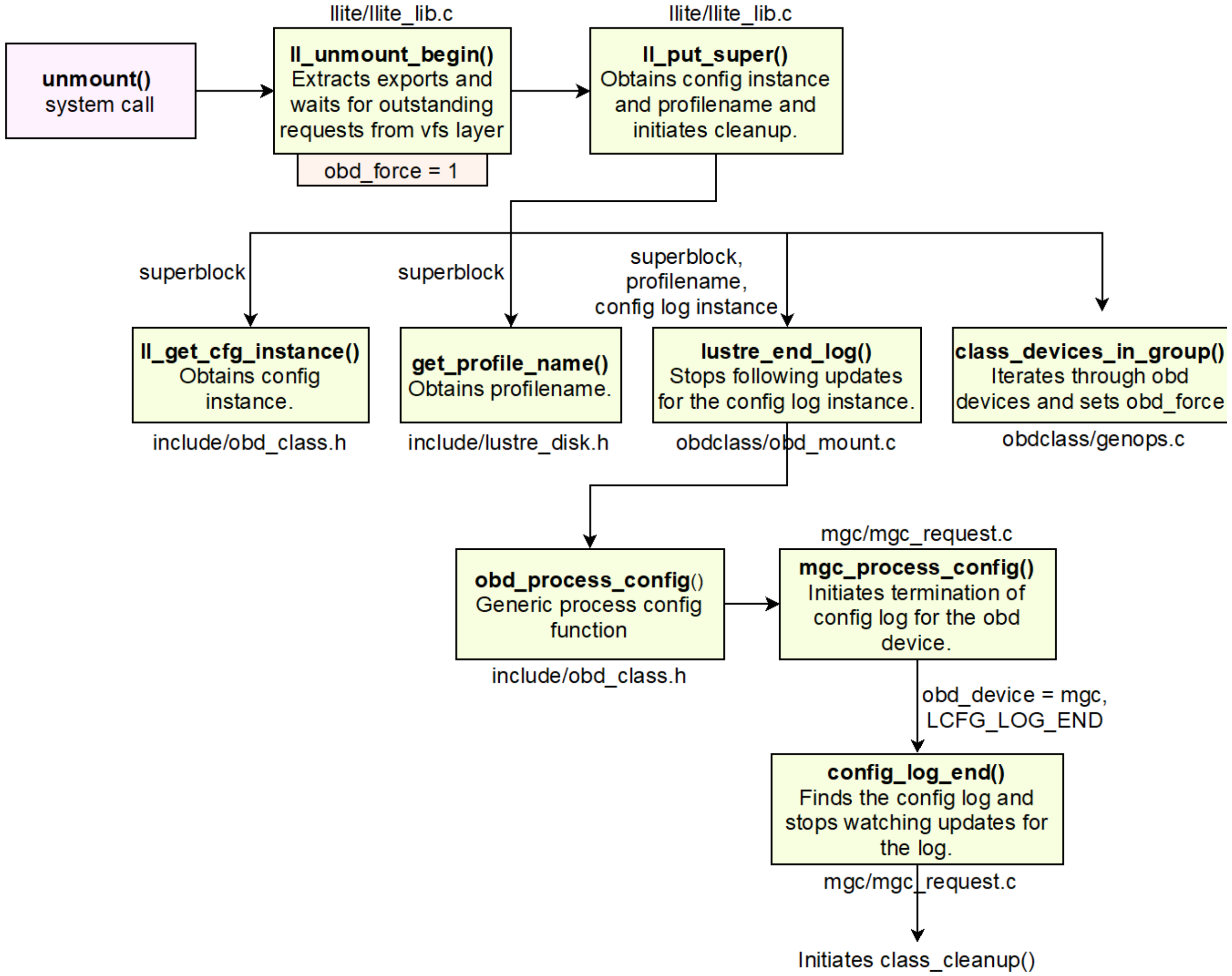
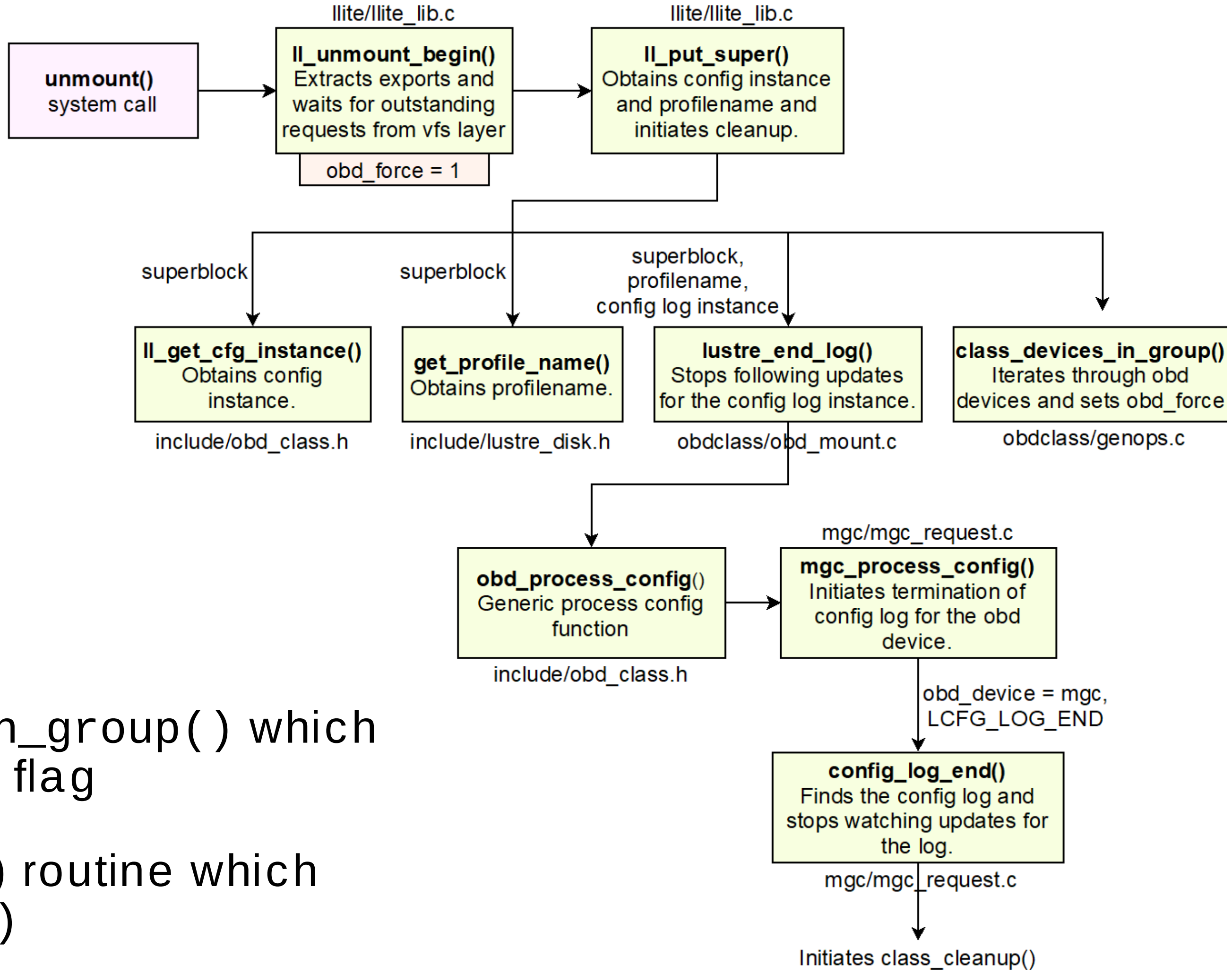

unmount()
system call
llite/llite_lib.c
ll_unmount_begin()
Extracts exports and waits for outstanding requests from vfs layer
obd_force = 1
llite/llite_lib.c
ll_put_super()
Obtains config instance and profilename and initiates cleanup.
superblock
superblock
superblock, profilename, config log instance
ll_get_cfg_instance()
Obtains config instance.
include/obd_class.h
get_profile_name()
Obtains profilename.
include/lustre_disk.h
lustre_end_log()
Stops following updates for the config log instance.
obdclass/obd_mount.c
class_devices_in_group()
Iterates through obd devices and sets obd_force
obdclass/genops.c
obd_process_config()
Generic process config function
include/obd_class.h
mgc/mgc_request.c
mgc_process_config()
Initiates termination of config log for the obd device.
obd_device = mgc, LCFG_LOG_END
config_log_end()
Finds the config log and stops watching updates for the log.
mgc/mgc_request.c
Initiates class_cleanup()

Figure 8. Lustre unmounting and initiation of class_cleanup() in obd device lifecycle [1]

Obd Device Life Cycle – `class_precleanup()` and `class_cleanup()`

- `ll_put_super()` obtains `cfg_instance` and profile name for super block using `ll_get_cfg_instance()` and `get_profile_name()`
- `lustre_end_log()` ensures to stop following updates for the config log
- `obd_process_config()` invokes device specific `mgc_process_config()`
- `config_log_end()` finds the config log and stop watching updates
- `ll_put_super()` invokes `class_devices_in_group()` which iterates through devices and sets `obd_force` flag
- Afterwards it calls `class_manual_cleanup()` routine which invokes obdclass functions `class_cleanup()` and `class_detach()`



unmount()
system call

llite/llite_lib.c
ll_unmount_begin()
Extracts exports and waits for outstanding requests from vfs layer
obd_force = 1

llite/llite_lib.c
ll_put_super()
Obtains config instance and profilename and initiates cleanup.

superblock
ll_get_cfg_instance()
Obtains config instance.
include/obd_class.h

superblock
get_profile_name()
Obtains profilename.
include/lustre_disk.h

superblock, profilename, config log instance
lustre_end_log()
Stops following updates for the config log instance.
obdclass/obd_mount.c

class_devices_in_group()
Iterates through obd devices and sets obd_force
obdclass/genops.c

obd_process_config()
Generic process config function
include/obd_class.h

mgc/mgc_request.c
mgc_process_config()
Initiates termination of config log for the obd device.

obd_device = mgc, LCFG_LOG_END

config_log_end()
Finds the config log and stops watching updates for the log.
mgc/mgc_request.c

Initiates class_cleanup()

Figure 8. Lustre unmounting and initiation of class_cleanup() in obd device lifecycle [1]

Obd Device Life Cycle – `class_precleanup()` and `class_cleanup()`

- `class_cleanup()` starts the shutdown of the obd device
- Sets `obd_stopping` flag to indicate cleanup has started
- Disconnects exports using `class_disconnect_exports()`
- Invokes generic function `obd_precleanup()` to ensure all exports are destroyed
- `class_cleanup()` destroys `uuid-export`, `nid-export`, and `nid-stats` hashtables
- `class_detach()` makes the `obd_attached` flag to zero
- `class_unregister_device()` unregisters the device
- `class_decref()` destroys last export by calling `class_unlink_export()`

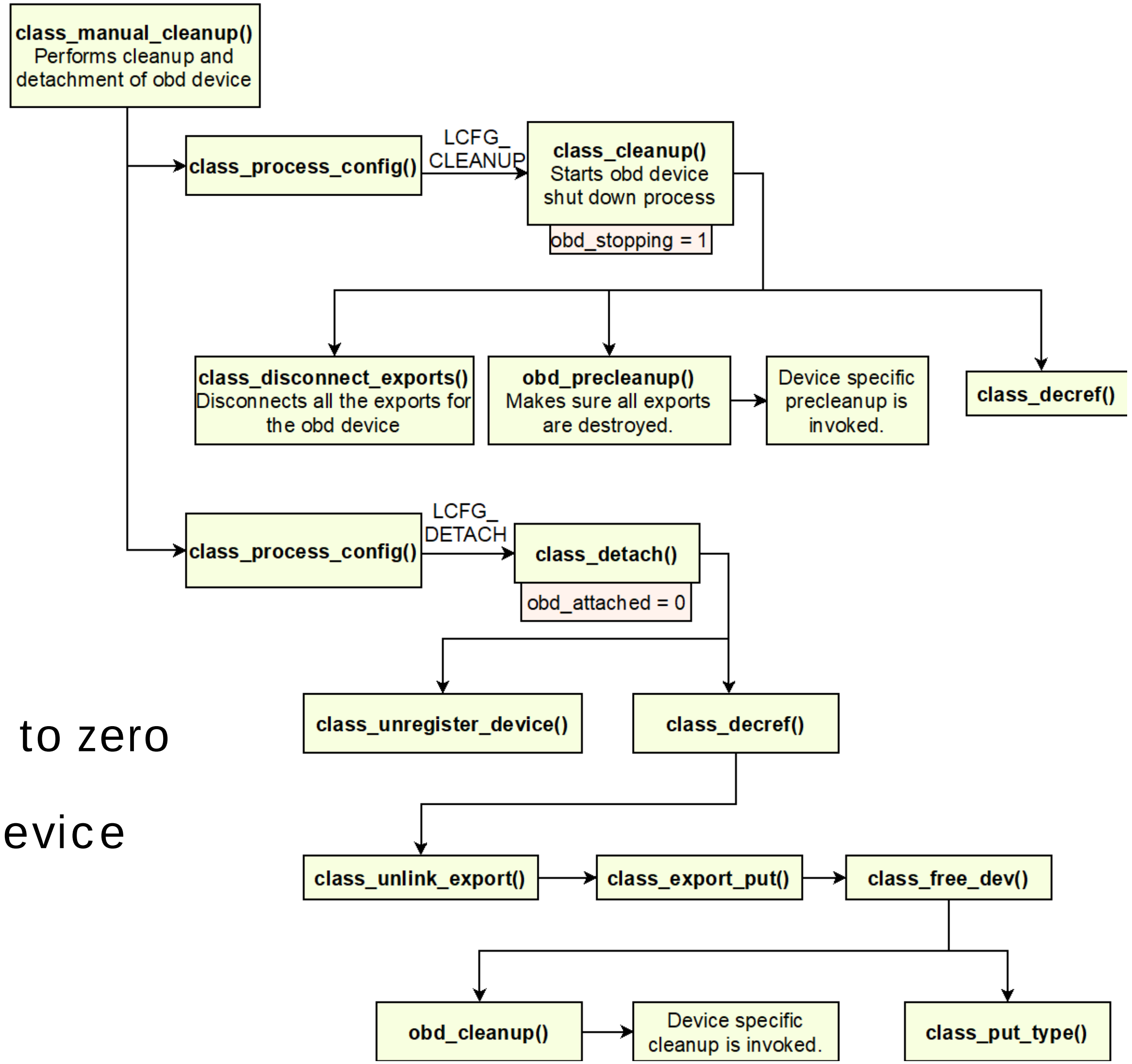

class_manual_cleanup()
Performs cleanup and detachment of obd device
class_process_config()
LCFG_
CLEANUP
class_cleanup()
Starts obd device shut down process
obd_stopping = 1
class_disconnect_exports()
Disconnects all the exports for the obd device
obd_precleanup()
Makes sure all exports are destroyed.
Device specific precleanup is invoked.
class_decref()
class_process_config()
LCFG_
DETACH
class_detach()
obd_attached = 0
class_unregister_device()
class_decref()
class_unlink_export()
class_export_put()
class_free_dev()
obd_cleanup()
Device specific cleanup is invoked.
class_put_type()

Figure 9. class_cleanup() workflow in obd device lifecycle [1]

OAK RIDGE National Laboratory

Obd Device Life Cycle – `class_precleanup()` and `class_cleanup()`

- `class_export_put()` frees the `obd device` using `class_free_dev()`
- `class_free_dev()` call device specific cleanup through `obd_cleanup()`
- `class_put_type()` unloads the module

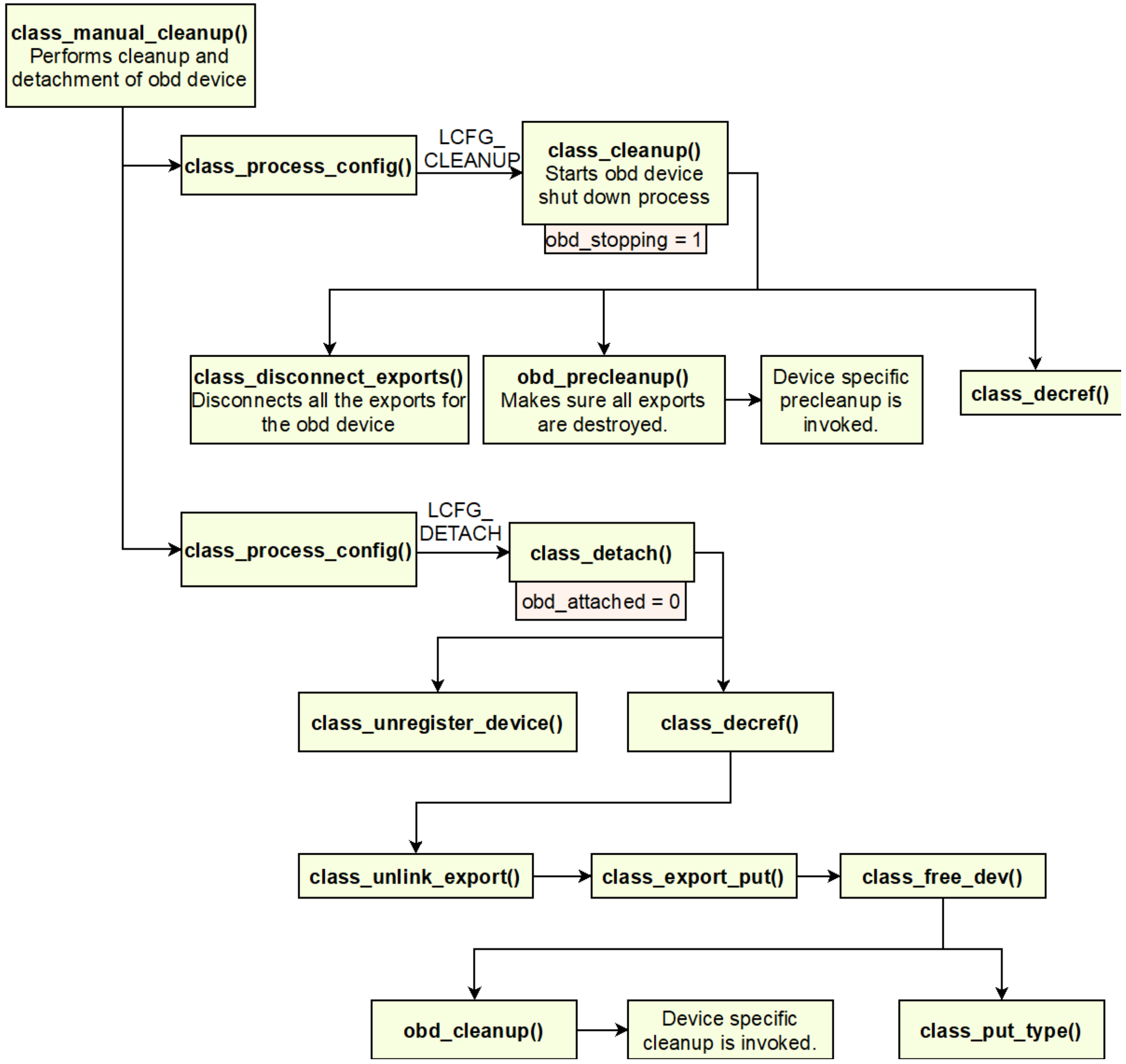

class_manual_cleanup()
Performs cleanup and detachment of obd device
class_process_config()
LCFG_CLEANUP
class_cleanup()
Starts obd device shut down process
obd_stopping = 1
class_disconnect_exports()
Disconnects all the exports for the obd device
obd_precleanup()
Makes sure all exports are destroyed.
Device specific precleanup is invoked.
class_decref()
class_process_config()
LCFG_DETACH
class_detach()
obd_attached = 0
class_unregister_device()
class_decref()
class_unlink_export()
class_export_put()
class_free_dev()
obd_cleanup()
Device specific cleanup is invoked.
class_put_type()

Figure 9. class_cleanup() workflow in obd device lifecycle [1]

Imports and Exports

- Lustre components like mdt and ost need one client obd device to establish communication between them
- Also applicable in case of Lustre client communicating with Lustre servers
- Client side obd device consists of exports and reverse imports

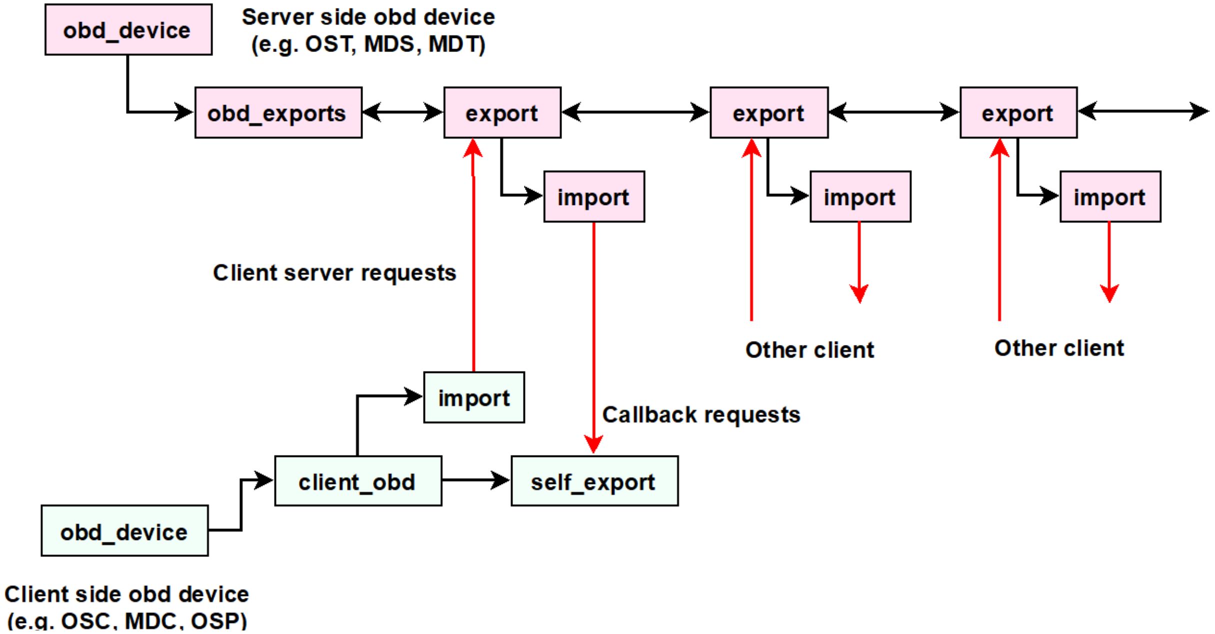


Figure 10. Import and export pair in Lustre [1]

- Client device sends request to server using its import and server receives it using its export
- Imports on server devices are called reverse imports because they send requests to client obd devices
- Client uses its self-export to receive these call backs requests from the server

Imports and Exports

- For any two obd devices to communicate with each other they need an import export pair
- `lctl dl` and `/sys/fs/lustre` directory show th obd devices
- Example of name of an obd device for communication between OST5 and MDT2 will be `MDT2-lwp-OST5`
- Client obd device that enables the communication is lwp
- lwp manages connections established from ost to mdt, and mdts to mdt0
- lwp is also used to send quota and FLD query requests
- Figure shows the communication between mdt and ost through osp client obd device

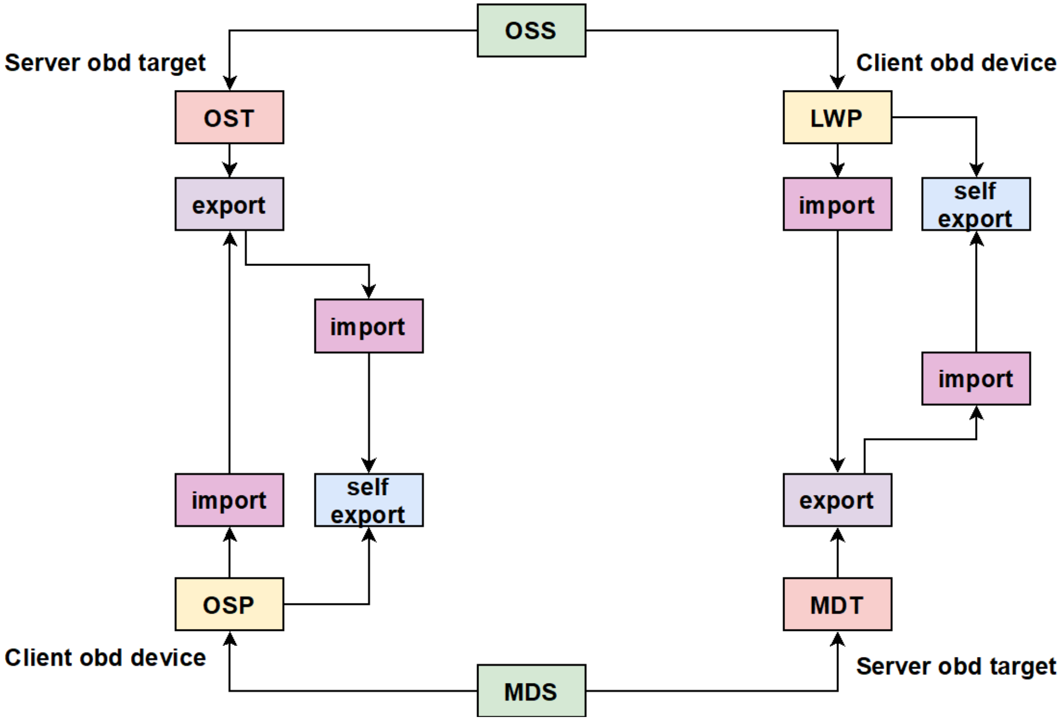


Figure 11. Communication between ost and mdt server obd devices in Lustre [1]

Imports and Exports

- obdfilter directory from `/proc/fs/lustre` lists osts present on the OSS node
- All osts have export connections listed in the nid format in their respective exports' directory
- Export connection information is stored in a file called export in each of the export connections directory
- Viewing the export file corresponding to MDT2 shows the following fields
  - `name`: Shows the name of the ost device
  - `client`: The nid of the client export connection
  - `connect_flags`: Flags representing various configurations for the lnet and ptl-rpc connections between the obd devices.
  - `connect_data`: Includes fields such as `flags, instance, target_version, mdt_index and target_index`
  - `export_flags`: Configuration flags for export connection
  - `grant`: Represents target specific export data

Useful APIs in Obdclass

- All obdclass related functions are declared in `include/obd_class.h` and definitions are found in `obdclass/genops.c`
- `class_newdev()` - Creates a new obd device, allocates and initializes it.
- `class_free_dev()` - Frees an obd device.
- `class_unregister_device()` - Unregisters an obd device by feeing its slot in obd_devs array.
- `class_register_device()` - Registers obd device by finding a free slot in in obd_devs array and filling it with the new obd device.
- `class_name2dev()` - Returns minor number corresponding to an obd device name.
- `class_name2obd()` - Returns pointer to an obd_device structure corresponding to the device name.
- `class_uuid2dev()` - Returns minor number of an obd device when uuid is provided.
- `class_uuid2obd()` - Returns obd_device structure pointer corresponding to a uuid.
- `class_num2obd()` - Returns obd_device structure corresponding to a minor number.

Acknowledgements

- More detailed information on the subsystems can be found in the wiki page "Understanding Lustre Internals"

- Thanks to Lustre team at ORNL – James Simmons, Rick Mohr and Sarp Oral

- This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

- References

1. George, Anjus, Mohr, Rick, Simmons, James, and Oral, Sarp. *Understanding Lustre Internals Second Edition*. United States: N. p., 2021. Web. doi:10.2172/1824954.