OAK RIDGE
NATIONAL LABORATORY

MANAGED BY UT-BATTELLE
FOR THE DEPARTMENT OF ENERGY

ORNL/TM-2009/117

Understanding Lustre Filesystem Internals

April 2009

Prepared by

Feiyi Wang
Sarp Oral
Galen Shipman
National Center for Computational Sciences

Oleg Drokin
Tom Wang
Isaac Huang
Sun Microsystems Inc.

UT-BATTELLE
ORNL-27 (4-00)

NATIONAL CENTER FOR
COMPUTATIONAL SCIENCES

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

ORNL/TM-2009/117

National Center for Computational Sciences

UNDERSTANDING LUSTRE FILESYSTEM INTERNALS

Feiyi Wang
Sarp Oral
Galen Shipman
Technology Integration Group
National Center for Computational Sciences

Oleg Drokin
Tom Wang
Isaac Huang
Lustre Center of Excellence
Sun Microsystems Inc.

Date Published: April 2009

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, Tennessee 37831-6283
managed by
UT-BATTELLE, LLC
for the
U.S. DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

Understanding Lustre Filesystem Internals

Abstract

Lustre was initiated and funded, almost a decade ago, by the U.S. Department of Energy Office of Science and National Nuclear Security Administration laboratories to address the need for an open source, highly scalable, high-performance parallel filesystem on then-present and future supercomputing platforms. Throughout the last decade, it was deployed over numerous medium- to large-scale supercomputing platforms and clusters, and it performed and met the expectations of the Lustre user community. At the time of this writing, according to the Top500 list, 15 of the top 30 supercomputers in the world use Lustre filesystem.

This report aims to present a streamlined overview of how Lustre works internally at reasonable detail including relevant data structures, APIs, protocols, and algorithms involved for the Lustre version 1.6 source code base. More important, the report attempts to explain how various components interconnect and function as a system. Portions of the report are based on discussions with Oak Ridge National Laboratory Lustre Center of Excellence team members, and portions of it are based on the authors' understanding of how the code works. We, the authors, bear all responsibility for errors and omissions in this document. We can only hope the report helps current and future Lustre users and Lustre code developers as much as it helped us understanding the Lustre source code and its internal workings.

Contents

List of Figures

1 Component View on Architecture

Lustre is a GNU General Public licensed, open-source distributed parallel filesystem developed and maintained by Sun Microsystems Inc. Due to the extremly scalable architecture of the Lustre filesystem, Lustre deployments are popular in scientific supercomputing, as well as in the oil and gas, manufacturing, rich media, and finance sectors. Lustre presents a POSIX interface to its clients with parallel access capabilities to the shared file objects. As of this writing, 15 of the top 30 fastest supercomputers in the world use Lustre filesystem for high-performance scratch space.

Lustre is an object-based filesystem. It is composed of three components: Metadata servers (MDSs), object storage servers (OSSs), and clients. Figure 1 illustrates the Lustre architecture. Lustre uses block devices for file data and metadata storages and each block device can be managed by only one Lustre service. The total data capacity of the Lustre filesystem is the sum of all individual OST capacities. Lustre clients access and concurrently use data through the standard POSIX I/O system calls.

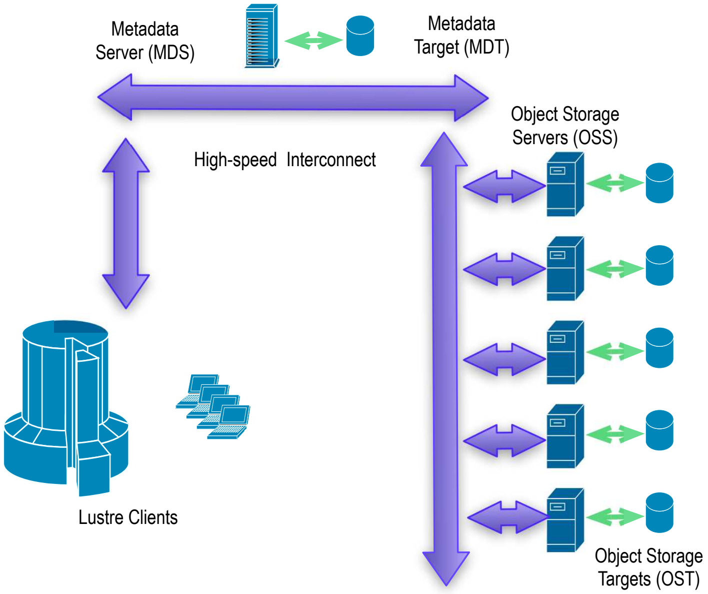

Figure 1: Lustre components.

- MDS (metadata servers) provides metadata services. Correspondingly, an MDC (metadata client) is a client of those services. One MDS per filesystem manages one metadata target (MDT). Each MDT stores file metadata, such as file names, directory structures, and access permissions.

- MGS (management server) serves configuration information of the Lustre filesystem.
- OSS (object storage server) exposes block devices and serves data. Correspondingly, OSC (object storage client) is client of the services. Each OSS manages one or more object storage targets (OSTs), and OSTs store file data objects.

The collection of MDS/MGS and OSS/OST are sometimes referred to as *Lustre server fronts*, fsfilt and ldiskfs as *Luster server backends*. In the following discussion, we start from the Lustre client side, and follow the general data and control thread all the way to the OST and MDS. The discussion touches on many components while skipping details to make the structural relationship more obvious.

Lustre Client

Lustre, being a POSIX-compliant filesystem, presents a unified filesystem interface such as open(), read(), write(), etc. to the user. In Linux, this unified interface is achieved through Virtual File System (VFS) layer (in BSD/Solaris, this would be known as `vnode` layer). There is a shim layer in Lustre called **llite** that is hooked with VFS to present that interface. The file operation requests that reach llite will then go through the whole Lustre software stack to access the Lustre filesystem, as shown in Figure 2.

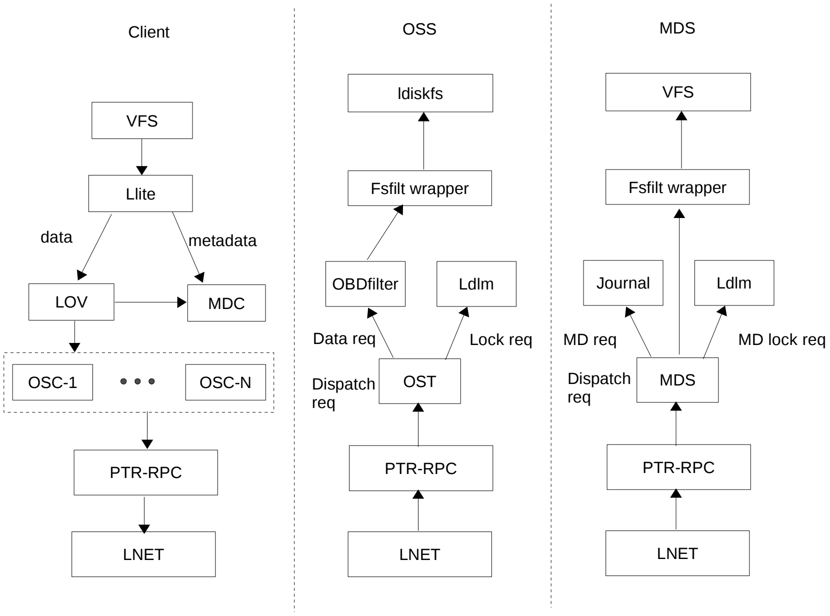

Figure 2: A component view of Lustre architecture.

In Lustre, general file operations such as create, open, read, etc. require metadata information stored on **MDS**. This service is accessed through a client interface module, known as **MDC**.

From the MDS point of view, each file is composed of multiple data objects striped on one or more OSTs. A file object's layout information is defined in the extended attribute (EA) of the inode. Essentially, EA describes the mapping between file object id and its corresponding OSTs. This information is also known as *striping EA*.

For example, if file A has a stripe count of three , then its EA might look like:

```
EA ---> <obj id x, ost p>
        <obj id y, ost q>
        <obj id z, ost r>
        stripe size and stripe width
```

So if the stripe size is 1MB, then this would means that [0,1M), [4M,5M) ... are stored as object *x*, which is on OST *p*; [1M, 2M), [5M, 6M) ... are stored as object y, which is on OST *q*; [2M,3M), [6M, 7M) ... are stored as object *z*, which is on OST *r*.

Before reading the file, client will query the **MDS** via MDC and be informed that it should talk to `<ost p, ost q, ost r>` for this operation. This information is structured in so-called LSM, and client side **LOV** (logical object volume) is to interpret this information so client can send requests to OSTs. Here again, the client communicates with OST through a client module interface known as OSC. Depending on the context, OSC can also be used to refer to an OSS client by itself.

All client/server communications in Lustre are coded as an RPC request and response. Within the Lustre source, this middle layer is known as Portal RPC, or **ptl-rpc**: It translates and interprets filesystem requests to and from the equivalent form of RPC request and response, and the **LNET** module to finally put that down onto the wire.

OSS

At the bottom of the OSS stack, are the familiar LNET and Portal-RPC layers. As with the client side stack, Portal RPC will interpret the request. The important thing to bear in mind is that the requests handled by OSS are data requests [1], not metadata requests. Metadata requests should be passed on and handled by MDS stack, as shown at the rightmost column in Figure 2.

Going up on the stack, **OST** acts like a dispatcher: it invokes different functions based on the type of request. Broadly speaking, there are two types of request: *lock* related and *data* related. The former will be passed onto **ldlm** (Lustre distributed lock manager) to handle and the latter will go to **obdfilter**. The obdfilter is a module that interconnects the Lustre stack and the regular OS stack, so to speak. It defines a generic API that translates a Lustre-specific request to the backend filesystem-specific request, with the help another wrapper API component called **fsfilt**. Conceptually, fsfilt is like a VFS layer, if you register proper file operations with it, it will use the particular filesystem as the backend; in the case of Lustre, this backend filesystem is currently **ldiskfs**. In the future, ZFS will be supported as backend filesystem as well, and fsfilt could probably be redesigned or replaced by a better filesystem-agnostic middle layer.

[1] As well as some size requests, such as glimpse requests from a client

MDS

The MDS software stack is similar to the OSS stack but, there are some differences between them. The main difference is that the MDS software stack does not have an obdfilter as can be seen in Figure 2. The dispatcher is called **MDS**. Of course, it does more than just dispatch, and it will be explained in detail in Section 6. For a metadata change request, MDS will do journaling a bit differently: it will start a transaction before directly invoking the VFS API. The particular component block may be called **dcache** as it mostly concerns operating on a **dentry** cache, but in a larger framework, it is really part of the VFS.

2 Lustre Lite

In this section, we describe how Lustre Lite connects and fits into Linux VFS, which is necessary for supporting VFS semantics and the POSIX interface. To summarize, Lustre Lite provides the following functions through a method table:

- Lustre specific file operations, through `ll_file_operations`.
- Lustre specific dentry operations, through `ll_d_ops` and its cache.
- Lustre specific directory operations, through `ll_dir_operations`.
- Lustre specific inode operations, through `ll_dir_inode_operations` and `ll_file_inode_operations`.
- Lustre specific file mapping operations, through `ll_file_vm_ops`.
- And many others, such as Lustre super block operations, address spaces, etc.

2.1 VFS Connection

Figure 3 presents an overall picture of how various data structures are connected. Some highlights are explained below.

A process maintains a list of associated open files. Each open file has an in-memory representation known as the `file` object. It stores the information necessary to interact between an open file and process. To the userland, this is presented by a file handle, `fd`. This data structure contains a field, `f_op`, that acts like a switchboard or pointer to a method table that provides functions specific to each filesystem. So a file read using system call `sys_read()` becomes:

```
file->f_op->read(...);
```

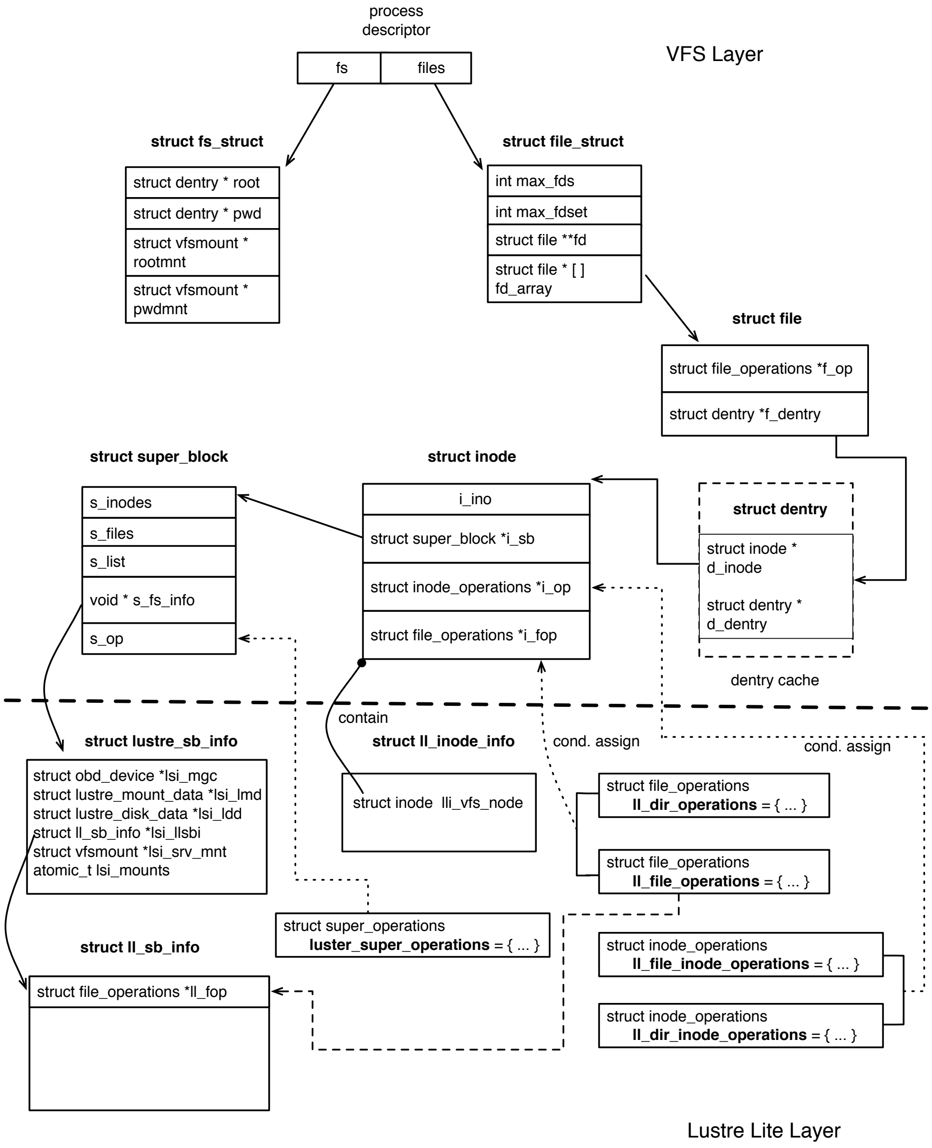
process descriptor
fs
files
VFS Layer
struct fs_struct
struct dentry * root
struct dentry * pwd
struct vfsmount * rootmnt
struct vfsmount * pwdmnt
struct file_struct
int max_fds
int max_fdset
struct file **fd
struct file * [ ] fd_array
struct file
struct file_operations *f_op
struct dentry *f_dentry
struct super_block
s_inodes
s_files
s_list
void * s_fs_info
s_op
struct inode
i_ino
struct super_block *i_sb
struct inode_operations *i_op
struct file_operations *i_fop
struct dentry
struct inode * d_inode
struct dentry * d_dentry
dentry cache
contain
cond. assign
cond. assign
struct lustre_sb_info
struct obd_device *lsi_mgc
struct lustre_mount_data *lsi_lmd
struct lustre_disk_data *lsi_ldd
struct ll_sb_info *lsi_llsbi
struct vfsmount *lsi_srv_mnt
atomic_t lsi_mounts
struct ll_inode_info
struct inode lli_vfs_node
struct file_operations ll_dir_operations = { ... }
struct file_operations ll_file_operations = { ... }
struct super_operations luster_super_operations = { ... }
struct ll_sb_info
struct file_operations *ll_fop
struct inode_operations ll_file_inode_operations = { ... }
struct inode_operations ll_dir_inode_operations = { ... }
Lustre Lite Layer

Figure 3: Hooking up Lustre with Linux VFS.

2.1.1 Dentry Object

Another important field defined in the `file` structure is `f_dentry`, which points to a dentry object (`struct dentry`) stored in a dentry cache, also known as the *dcache*. Essentially, VFS will create a dentry object the first time a file or directory is about to be accessed. If this is a non-existent file/directory, then a negative dentry will be created. As an example, take the following pathname: /home/bob/research08; it is composed of four path components: /, home, bob, and research08. Correspondingly, the path lookup will create four dentry objects for each component. Each dentry object associates *the* respective component with its inode through field d_inode.

The inode object stores information about a specific file, uniquely identified by an inode number. ULK3[2] has exhaustive listings of field definitions for inode structure. What is important to know is that (1) i_sb points to the VFS superblock; (2) i_op is the switchboard for inode operations such as:

```
create(dir, dentry, mode, nameidata)
mkdir(dir, dentry, mode)
lookup(dir,dentry,nameidata)
...
```

The first method creates a new disk inode for a regular file, associated with a dentry object in some directory. The second method creates a new inode for a directory associated with some dentry object in some directory. And the third one searches a directory for an inode corresponding to a filename included in a dentry object.

2.1.2 Lustre Superblock

VFS layer defines a generic superblock object (struct super_block), which stores information about the mounted filesystem. One particular field s_fs_info points to superblock information that belongs to a specific filesystem. In the case of Lustre, this filesystem specific data is represented by the structure lustre_sb_info, which stores information needed for mounting and unmounting Lustre filesystem. It further connects to another structure ll_sb_info, which contains more Lustre Lite[3] specific information about filesystem state only for clients.

Lustre specific superblock operation is defined in struct variable lustre_super_operations. [4] The initialization of *correct* superblock operations takes place when we initially establish the in-memory superblock data structure in function client_common_fill_super():

```
sb->s_op = &lustre_super_operations;
```

It is worth mentioning that when creating Lustre files, the alloc_inode() superblock method is implemented by ll_alloc_inode() function. It will create a Lustre specific inode object ll_inode_info and return the VFS inode embedded in it. Pay attention to the particular way in which the generic VFS inode and Lustre inode interconnect with each other: this method creates and fills the VFS inode structure along with extra state information the Lustre filesystem needs, but it only returns the VFS inode structure &lli->lli_vfs_inode embedded in lli. [5]

[2] Understanding the Linux Kernel, the third edition.

[3] We use Lustre Lite and llite interchangeably.

[4] See more details in lustre/llite/super25.c, which is actually for kernel 2.6

[5] The fact that we allocate one big structure that holds both VFS inode and Lustre private state information is an implementation detail, and it is not necessary to be this way. It used to be that private state info was allocated separately and a pointer from VFS inode was used to access it.

```
    static struct inode **ll_alloc_inode(struct super_block *sb)
    {
        struct ll_inode_info *lli;
        ...
        return &lli->lli_vfs_inode;
    }
```

To retrieve the parent structure from a child structure, Lustre defines a helper method `ll_i2info()`, which essentially invokes kernel macro `container_of` by:

```
    /* parameters of macro: ptr, type, memeber */
    return container_of(inode, struct ll_inode_info, lli_vfs_inode);
```

2.1.3 Lustre inode

The initialization of correct inode and file/dir functions takes place when inode is being filled in during `ll_read_inode2()`. There are four struct variables defined for it, and two of these are for inode operations: `ll_file_inode_operations` and `ll_dir_inode_operations`. Two of these are for file/directory operations, `ll_file_operations`, and `ll_dir_operations`. In each case, file and directory each has its own set, and which to assign depends on the inode or the file itself. [6] The following snippet shows an example of a definition for file operations:

```
/* in dir.c */
struct file_operations ll_dir_operations = {
        .open     = ll_file_open,
        .release  = ll_file_release,
        .read     = generic_read_dir,
        .readdir  = ll_readdir,
        .ioctl    = ll_dir_ioctl, ...
};
/* in file.c */
struct file_operations ll_file_operations = {
        .read        = ll_file_read,
        .write        = ll_file_write,
        .open        = ll_file_open, ...
```

For example, if the inode to be created is a file, then `i_fop` will be assigned as `ll_file_operations`; if the inode to be created is a directory, then `i_fop` will be assigned as `ll_dir_operations`:

```
if (S_ISREG(inode->i_mode)) {
    inode->i_op = &ll_file_inode_operations;
    inode->i_fop = sbi->ll_fop;
    ...
} else if (S_ISDIR(inode->i_mode)) {
    inode->i_op = &ll_dir_inode_operations;
    inode->i_fop = &ll_dir_operations;
    ...
```

A general observation on the pattern: the pointer for the method table is initialized and established properly by the party or the function that is creating the new instance of it.

[6] These variables are scattered in various places such as `file.c`, `dir.c`, `namei.c`.

2.2 Path Lookup

Path lookup is a relatively complex task and one of the most important and frequently performed. Linux VFS does most of the heavy lifting; we want to emphasize the junction at which Lustre does specific things. In this section, we outline basic steps with enough details to follow the main thread of the code but skip many branches one has to take care of in real code, such as:

- the path name contains component of `.` and `..`,
- the path name contains symbolic links, which may induce circular reference,
- the access rights and permission check,
- the path name contains mount point of another filesystem,
- the path name contains not-yet-existing file,
- `LOOKUP_PARENT` flag set,
- the path name does not have a trailing slash.

Lookup can be invoked from `sys_open()` call, the usual call path from there is to do `filp_open()` and `open_namei()`. It is this last function that initiates `path_lookup()` call. In particular, if a file is opened with `O_CREAT` flag in the parameter for access mode, the lookup operation will be set with `LOOKUP_PARENT`, `LOOKUP_OPEN`, and `LOOKUP_CREATE`. The end result of path lookup is either:

- return dentry object of last path component if it exists; or
- return dentry object of next-to-last path component if it does not, as in the case of creating a new file. From there, you can allocate a new disk inode by invoking the `create` method of the parent inode.

Now we focus on the lookup specifics. If the path starts with `/`, then it is an absolute path: search starts with the process root directory in `current->fs->root`. Otherwise, a search starts with `current->fs->pwd`. We also know the dentry object as well as its inode of beginning directory at this point (refer back to figure 3 to see why). The `nameidata` keeps track of the last resolved path component with `dentry` and `mnt` fields. Initially, they are assigned with *starting directory*. The core lookup operation is performed by `link_path_walk(name, nd)`, where `name` is the path name, `nd` is the address of the `nameidata` structure.

1. Consider next component to be resolved; from its name, compute 32-bit hash value to be used when looking in the dentry cache hash table.
2. Set `LOOKUP_CONTINUE` flags in `nd->flags` to denote that there are more components to be analyzed.

3. Invoke `do_lookup()` to search dentry object for the path component. If found (skip revalidation), then this path component has been resolved, and move on. If not found, then invoke `real_lookup()`. At the end of the cycle, the `dentry` and `mnt` field of local dentry variable `next` will point to, respectively, the dentry object and the mounted filesystem object of the path component we attempt to resolve.

4. If the above `do_lookup()` reaches the last component of the pathname, and assuming that it is not a symbolic link as assumed at the beginning, then this is our destination. All we need to do is to store the dentry object and mnt info in the passed nameidata `nd` and return without error:

   ```
   nd->dentry = next->dentry;
   nd->mnt  = next->mnt;
   ```

Lustre specific operation is handled in the `real_lookup()` call. A dentry is created by VFS and is passed into the filesystem specific lookup function. Lustre's lookup responsibility is to locate or create a corresponding inode and fill it with correct information. If inode could not be found, then the dentry still remains, only it has a `NULL` inode pointer. Such dentries are called *negative* meaning there is no such file with this name. The particular code segment for this switching is given below.

```
struct dentry *result;
struct inode *dir = parent->d_inode;
...
result = d_lookup(parent, name);
if (!result) {
    struct dentry *dentry = d_alloc(parent, name, nd);
    if (dentry) {
        result = dir->i_op->lookup(dir,dentry,nd);
        ...
 }
```

Now the lookup is passed on Lustre's side, and follow-on operations might involve contacting MDS for more information.

There is also a `cached_lookup` path that calls in a `->revalidate` method provided by the Lustre. This method verifies that the cached dentry/inode is still valid and does not need to be updated from the server.

2.3 I/O Path

This section starts with discussions on three I/O paths traveled by Lustre: Async I/O, Group I/O, and Direct I/O. Then we discuss how Lustre interfaces with VFS by surrendering control of I/O (in most cases) to VFS, which does more preparations and then reads/writes data on a page-by-page basis through address space methods, which are in turn provided by Lustre. So think of this as a process of: VFS to llite through hooks, llite then calls into VFS for processing help, and VFS then hands control back to llite – an in and out intertwined process.

2.3.1 Asynchronous I/O

This is probably the most traveled I/O path in Lustre, and we will provide a top-down description on the write process. Read operation is very similar. The registered file write operations for VFS are discussed in Sec. 2.1.

1. The `writev()` is for older kernels, newer ones will use `aio_write()`. Our entry point of analysis is `ll_file_write()`. This function defines a `iovec` with base pointing to the buffer provided at the user space, and length is the number of bytes to write.

   ```
   struct iovec local_iov = { .iov_base = (void __user *) buf,
                              .iov_len  = count };
   ```

   Another structure initialized here is `kiocb`, which records the status of the I/O. Passing in as parameters both `iovec` and `kiocb`, it invokes either `ll_file_writev()` or `ll_file_aio_write()` depending on the kernel version. In our case, we follow the latter. Codewise, both functions are implemented as one, only with slightly different prototype declarations:

   ```
   #ifdef HAVE_FILE_WRITEV
   static sszie_t ll_file_writev(
                               struct file *file,
                               const struct iovect *iov,
                               unsigned long nr_segs,
                               loff_t *ppos) {
   #else /*AIO stuff */
   static ssize_t ll_file_aio_write(
                               struct kiocb *iocb,
                               const struct iovec *iov,
                               unsigned long nr_segs,
                               loff_t pos)
   {
       struct file *file = iocb->ki_filp;
       loff_t *ppos = &iocb->ki_pos;
   #endif
   ```

2. The key to understanding this function (`ll_file_aio_write()`) is that Lustre breaks the write into multiple chunks based on the stripe size, then asks for a lock on each chunk in a loop. [7] This is to avoid complications when you have to ask a lock on a large extents. Although we said earlier that LOV is the layer that handles stripe info, Lustre Lite is very much aware of that as shown in this case – you can see how closely coupled they are. The stripe info is obtained by:

   ```
   struct lov_stripe_md *lsm = ll_l2inof(inode)->lli_smd;
   ```

   Lustre controls the size of each write by setting up a copy of the original iov control structure, `iov_copy`, then goes back and asks one of common routines in the kernel to drive the write:

   ```
   retval = generic_file_aio_write(iocb, iov_copy, nrsegs_copy, *pppos);
   ```

[7] The lock granularity is page unit, which is useful when you do small I/O. Normally, you would request a lock on stripe size. One exception is if it is `O_APPEND` write, then the lock is for the entire file.

We tally the number of bytes to write in *iov and use count to track the number of bytes remaining to be written. This is repeated until an error is hit or all bytes are written.

Before we make a generic write routine, in addition to getting a lock by calling ll_file_get_tree_lock_lov(), there are a few corner cases this function needs to deal with:

- If user-application opens the file with O_LOV_DELAY_CREATE, but starts to write without first calling ioctl to set it up, then we need to fail this call.
- If user-application opens the file with O_APPEND, then we need to get a lock on the entire content: lock_end is set with OBD_OBJECT_EOF, or -1 to represent the end of the file.

3. generic_file_aio_write() is just a wrapper for __generic_file_aio_write_nolock(); both functions have been described in ULK3, and we will not dwell on it here. Since direct I/O is not covered in this section, the write flow goes into generic_file_buffered_write(). This is where the write is further broken into pages, and page cache is allocated. The main body of work is performed in the loop given below.

```
do {
    ...
    status = a_ops->prepare_write(file, page, offset, offset+bytes);
    ...
    copied = filemap_copy_from_user(page, offset, buf, bytes);
    ...
    status = a_ops->commit_write(file, page, offset, offset+bytes);
    ...
} while (count);
```

First, we prepare the page write through the Lustre registered method. The preparation involves checking if the starting position is aligned at the beginning and if it needs to be read from disk first (of course, there is no local disk operation here in Lustre, but VFS sees it that way). Then it copies a page worth of data from user space to the kernel. Finally, we once again ask Lustre specific method to write the page out. So the control is passed in and out of Lustre code twice.

There are a few interesting points to make on page and boundary management. Let's say the logical file position for write is 8193 and the page size is 4KB. As this is a page-based write (both prepare write and commit write are page-based), it first calculates page index (2) and page offset (1), and bytes are the maximum bytes you can write in that page. However, if the remaining count is less than that, we need to adjust it by the exact number of bytes we will write for this page.

```
index = pos >> PAGE_CACHE_SHIFT;
offset = (pos & (PAGE_CACHE_SIZE -1));
bytes = PAGE_CACHE_SIZE - offset;
bytes = min(bytes, count);
```

This calculated logical page index will be used to locate or allocate a page in the page cache and be associated with this file mapping:

```
struct address_space *mapping = file->f_mapping;
page = __grab_cache_page(mapping, index, &cached_page, &lru_pvec);
```

A small digression: The `__grab_cache_page()` is a helper function that is only used for generic file write requests. The basic flow of this function is to first check if such a page already exists in the pagecache and return it if so. Otherwise, it allocates a new page by invoking `page_cache_alloc()` and adds it to page cache (`add_to_page_cache()`). However, another thread might allocate a page at this offset just between the last check and now, so adding to pagecache can fail. In that case, we need to check and retrieve the page from pagecache again:

```
repeat:
    page = find_lock_page(mapping, index);
    if (!page) {
        ... page_cache_alloc(mapping);
        err = add_to_page_cache(*cached page, mapping, index, GFP_KERNEL);
        if (err == -EEXIST)
            goto repeat;
```

A small optimization is made here at the cost of polluting the code: if a just-allocated page cannot be added to the page cache as shown, instead of returning it, it keeps it in `cached_page`, so next time when a new page is requested, we don't have to call the allocation function again.

4. Now we can move to prepare write. It is declared as given below.

```
int ll_prepare_write(struct file *file, struct page *page,
                     unsigned from, unsigned to)
```

The `ll_prepare_write()` is invoked with `from` set as the `offset` and `to` set as the `offset + bytes`. This is exactly the boundary issue we discussed above. Overall, this method is to ensure that *the page is up to date*, and if this is not a full page write, then read in the partial page first.

A few structs are used in this function; their meanings are as follows:

`struct obdo` is for *on the wire* representation of a Lustre object and its related information.

`strut brw_page` is for describing the state of the page for sending.

`struct obd_info` is for passing parameters between Lustre layers.

Also in this method, we need to check the end of file (EOF) for cases like **partial page write**: if EOF falls within the page (write beyond EOF), then we need to fill in non-overwriting portions with zeros; otherwise, we need to preread it.

5. Next, the LOV initiates preparing page by `lov_prep_async_page()`.

There are three structs defined at each layer a page write needs to go through. At the Lustre Lite layer, there is the `ll_async_page` (LLAP). The LOV defines the `lov_async_page` (LAP) and the `osc_async_page` (OAP).

6. If the write operation is async, it will be handled by `osc_queue_async_io()`, and internally it calls function `osc_enter_cache()`, which does page accounting to ensure we do not have more dirty pages than allowed, and blocks on attempts to add extra dirty pages. It also enforces grants in similar manner.

   Therefore, in (at least) two conditions an OAP will not be able to go into the cache:

   - If the size of the cache is less than 32MB, then it is O.K. to put OAP into cache, return 0; otherwise, return an error code.
   - If the grant is not sufficient on the client side, then again return an error code. Upon each OSC client's initialization, one can assume that the OST server side granted the client some space. A **grant** is just a number promised by the server that this client can flush out this much data and knowing the server can deal with it. The initial grant is 2MB, which is two RPC's worth of data. Then, write request is issued by the client, and it can ask for more grant if needed. At the same time, along with each data transfer or write, client needs to keep track of this grant and not to overrun it.

   If the page is unable to get into the buffers, a group I/O or synchronous I/O will be tried later.

7. After the OAP caching check is done, `loi_list_maint()` is called to put it on the proper list, and readying it for either read or write actions.

8. Function `osc_check_rpcs()` builds RPCs for each object in `lov_oinfo`. Note that each RPC can carry content for only one data object.

2.3.2 Group I/O (or Synchronous I/O)

Group I/O is triggered when an OAP page cannot be successfully put into cache, most likely because of an insufficient grant. In that case, Lustre Lite will create a structure `obd_to_group` to hold OAP. This page will then be added to the client obd's ready list, with the URGENT flag set. Finally, `oig_wait()` would be invoked and wait for group I/O to finish.

Notice that group I/O waits on the operation; therefore, it is also known as **synchronous** I/O. Second, all group I/O is urgent I/O as well as read operations. In contrast, in **asynchronous** I/O, the OAP cache (when admitted) will go into the write list.

It is also worth noting that except direct and lockless I/O, all reads are done as group I/O. Direct I/O is briefly discussed below. Lockless I/O is a special kind of I/O (now disabled except in liblustre) where clients do not get any locks but instead instructs the server to take the locks on the client's behalf.

2.3.3 Direct I/O

For direct I/O, VFS passes an `iovec`, which describes a segment of data to transfer. It is simply defined by a start address `void * iobase` and a size, `iov_len`. Direct I/O requires that the start address be page-aligned.

Lustre Lite operates on each page and invokes `ll_direct_IO_26()` and `obd_brw_async()`, and this essentially gets converted to `osc_brw_async()` calls that later will be used for building RPC requests.

2.3.4 Interface with VFS

At the top level of VFS in general and Lustre Lite in particular, there are several places where you can register your own read and write methods, all through address space operation struct. For read operations, it is `readpage()` and its the vector version of `readpages()`. For write, it is a bit more complicated. The first entry point is `writepage()`, which can be triggered by:

- When memory pressure crosses the threshold, VM will trigger the flushing of dirty pages.
- When a user application does an `fsync()` and forces the flushing of dirty pages.
- When a kernel thread does the periodical flushing of dirty pages.
- When the lock on the page is revoked (block request) by the Lustre lock manager, the dirty page therefore must be flushed.

The implementation of `readpage()` and `writepage()` are optional, and not all filesystems support them. However, these methods are needed to take advantage of default read/write actors in kernel (*i.e.*, the generic implementation of `do_generic_file_read()` and `do_generic_file_write()`). It simplifies the interactions with kernel cache and VM. Also, both methods are needed to provide `mmap` support.

Filesystem registers the functions `prepare_write()` and `commit_write()` via an address space object as well.[8] So, an address space object can be characterized as the bridge from file space to storage space.

The address space operation in Lustre is defined in `lustre/llite/rw26.c`:

```
struct address_space_operations ll_aops = {
    .readpage           =   ll_readpage,
    .direct_IO          =   ll_direct_IO_26,
    .writepage          =   ll_writepage_26,
    .set_page_dirty     =   __set_page_dirty_nobuffers,
    .sync_page          =   NULL,
    .prepare_write      =   ll_prepare_write,
    .commit_write       =   ll_commit_write,
    .invalidatepage     =   ll_invalidatepage,
    .releasepage        =   ll_releasepage,
    .bmap               =   NULL
};
```

There is an `*f_mapping` field in `file` object pointing to the address space object associated with the file. The connection is established at file object creation.

[8] Linux Kernel 2.6 introduces `write_begin()` and `write_end()`

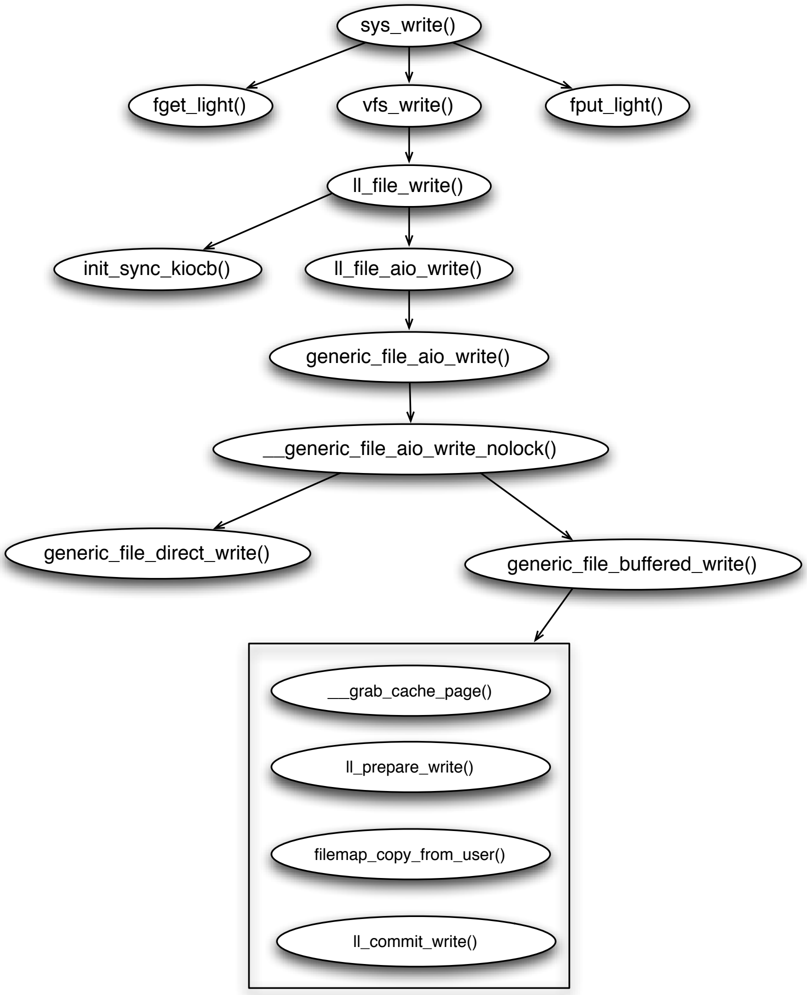

Figure 4: System call graph on the write path as in Linux kernel 2.6.22-14.

2.4 Read-Ahead

Lustre client read-ahead happens in read-page and is controlled by a structure, `ll_readahead_state`, which is defined in `lustre/llite/llite_internal.h`. This is a per-file structure and encodes the following information:

- **Read history** (a) how many contiguous reads have happened; (b) if it is stride read mode, then how many contiguous stride reads have happened; and (c) stride gap and stride length.

- **Read-ahead** windows (i.e., read-ahead window starting and end points). The more contiguous read happens, the longer the read-ahead window grows.
- State information that helps to **detect read-ahead pattern**.

The read-ahead algorithm is described below where each client could read-ahead a maximum of 40MB:

1. In read-page (`ll_readpage`), the read-ahead state of the file will be updated according to the current status.
   (a) If this page offset is located in a contiguous window (in +8, -8 window with the last page), then `ras_consecutive_requests` (defined by `ll_readahead_state`) will increase by 1. If it is the first page of this read, the read-ahead window length will increase by 1MB. So if the read is contiguous, the read-ahead window increases with an increasing number of read opeartions.
   (b) If this page is not inside the contiguous window, then it will check whether it is in stride-read mode. In this case, it will compare current stride length/stride gap with past history. If they are equal, then `ras_consecutive_stride_requests` will be increased by 1. If it is the first page of this read, stride read-ahead window will also increase by 1MB.
   (c) If page is neither in contiguous window nor in stride-contiguous window, then all the read-ahead state will be reset. For example, `ras_consecutive_pages` and `ras_consecutive_requests` will be reset to 0.
2. Next, read page will do real read-ahead according to the state updated in the previous step.
   (a) Increase the read-ahead window and try to cover all the pages for this read. This explains why large reads perform better than small reads.
   (b) Adjust the read-ahead window according to the real file length and calculate how many pages will be read in this read-ahead.
   (c) Do actual read-ahead.

There is a read-ahead stats file in the proc (`/proc/fs/lustre/llite/XXXXX/read_ahead_stats`) where states of read ahead can be viewed.

3 LOV and OSC

From a high level perspective, the job of LOV is to direct pages to the correct OSCs and OSC's job is to assemble a vector of dirty pages, group them, and send them to OST over the wire (of course, through the Portal RPC and LNET).

3.1 OBD Device Operations

One of the general code organization patterns or implementation techniques employed and that can be seen across LOV, OSC, and MDC are *obd device oriented operations*. An obd device is characterized by a method table defined by `struct obd_ops`, somewhat similar to the aforementioned VFS file, inode, dentry operations. The idea is that you can be insulated from knowing the exact obd devices you are talking to and use the generic wrapper method prefixed by `OBD_` instead. Let's take a glimpse of the methods this table defines:

```
struct obd_ops {
    struct module * o_owner;
    int (o_setup)(struct obd_device *dev, obd_count len, void *data);
    int (o_attach)(struct obd_device *dev, obd_count len, void *data);
    int (o_detach)(struct obd_device *dev);
    int (o_setattr)(struct obd_export *exp, struct obd_info *oinfo ...);
    ...
}
```

Although there are over 70 methods defined in the structure, only a small set of methods are supported across the board. Therefore, it has lost the generic character that was originally envisioned. Nonetheless, this method table provides a roadmap for us to explore activities around the particular obd devices.

For LOV, another pattern to note is that this is the layer that interprets stripe information; therefore, a file operation here often becomes an operation on a set of objects.

When LOV module initializes, it registers the obd device operations it supports to a class type manager. Essentially, this class type manager manages a class name and its associated properties in a flat space.

```
rc = class_register_type(&lov_obd_ops, lvars.module_vars, LUSTRE_LOV_NAME);
```

3.2 Page Management

Page management is an activity across multiple layers. Page cache is a generic memory space maintained by the kernel. Lustre pages are just those special pages the Lustre system is aware of. The special properties are stored in the private field of a page descriptor. It is divided into three portions: `llap`, `lap` and `oap`. The responsible layer that will do manipulation on its portion are `llite`, `lov` and `osc`, correspondingly, as illustrated below:

```
<-- llite --> <-- lov --> <-- osc  -->
+------------+-----------+-----------+
|    llap    |    lap    |    oap    |
+------------+-----------+-----------+
```

To keep this discussion more tangible, we will refer to the following use case from time to time to keep us focused: A user space application wants to create file *A*, then write 6.5MB of data to it. The stripe size is 1MB, stripe width is 3, and OSTs are numbered as OST_1, OST_2, and OST_3. The file A's data objects are named A_1, A_2, and A_3, correspondingly. The other files on these particular OSTs will be called B_1, B_2, B_3 and C_1, C_2, C_3, as illustrated in Figure 5. At this point, we need to define or clarify a few concepts and data structures. At VFS and Lustre Lite layers, the read and write happens on the page unit – note that it has no bearing on the actual data transfer done by Portal RPC and LNET.

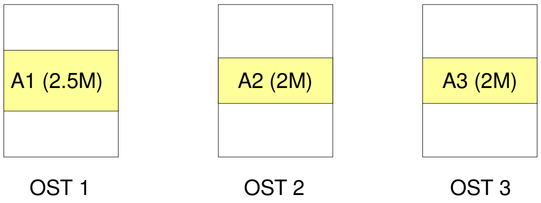

Figure 5: Lustre Lite object placement example.

A page of data needs to find its home on a file data object on the OST disk. The location information is essentially a tuple of <*OST index*, *offset within file data object in page unit*, *offset within page*>. Here is how it is defined:

```
struct lov_async_page {
    int         lap_stripe;     /* same as OST logical index */
    obd_id      lap_loi_id;     /* object id as seen by OST */
    obd_off     lap_sub_offset; /* offset within object in page-unit */
    ...
}
struct osc_async_page {
    obd_off     oap_obj_off;    /* offset within object in page unit */
    unsigned    oap_page_off;   /* offset within a page */
    ...
}
```

3.3 From OSC Client To OST

To interact with an OST, the client side needs to create a corresponding OSC client structure, client_obd. This is a one-to-one mapping. Note that client_obd represents other clients such as MGC, MDC as well. So in that sense, it is fairly generic and some fields may only make sense for a particular client.

For each data object handled by OSC client, there is one lov_oinfo object (loi) created, and each describes an object associated the OST. Each lov_oinfo further links to OAP pages for both read and write.

The lov_oinfo struct has a link back to client_odb, therefore, it can be used as the starting point, to search all lov_oinfo for each object this client is processing and to continue to locate all OAP pages that have been cached dirty during the write operation.

Figure 6 illustrates the case where two OSC clients interact with two OSTs. The first OST holds data objects of A_1, B_2, and C_3; the second OST holds A_2, B_1.

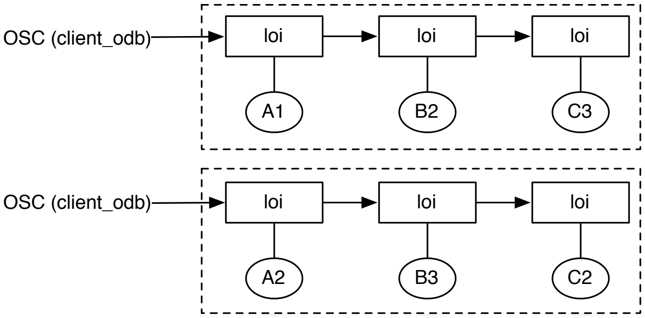

Figure 6: Connecting OSC client, `lov_oinfo` and OST objects.

3.4 Grant

Each client buffers some amount of dirty data. Currently, the default setting for each OSC is 32 MB.[9] Considering the number of clients we can have in the system, it is easy to see that when all clients flush their dirty buffer, the server side might run out of free space on the filesystem. To avoid this situation, the server establishes a certain limit on the amount of dirty data it can flush or transfer for each client, known as the *grant*. Clients are not allowed to have more dirty data than the grant. Clients can ask for more, and the server exercises the discretion to increase the grant and by how much. It is adjusted according to the amount of dirty data the client is caching. Every time a client dirties a page, it is subtracted from that grant. When grant reaches zero, all I/O becomes synchronous until the client can increase its grant.

4 LDLM: Lock Manager

The basic design idea of Lustre Lock Manager comes from VAX DLM. There are some fundamental concepts we need to explain before we can dive into the code and understand how it works.

4.1 Namespace

The first concept we will cover is the *namespace*. Whenever you request a lock, you are asking a lock for a certain resource in a namespace, and there is one namespace defined per Lustre service. To put this in a practical context, say your Lustre system is composed of ten OSTs, then from LDLM point of view, there are ten namespaces. Furthermore, MDS and MGS each have their own namespaces here. A namespace in Lustre is defined by struct `ldlm_namespace`. It has a reasonable amount of comments for the fields in the source code, so here we just focus on a few of the less-obvious ones.

[9] This is tunable through proc filesystem.

Type	Field	Description
ldlm_side_t	ns_client	is this a client-side lock tree?
struct list_head	*ns_hash	hash table for all resources in this namespace
__u32	ns_refcount	number of resources in the namespace
struct list_head	ns_list_chain	circular list of all namespaces
struct list_head	ns_unused_list	resources without any locks
int	ns_nr_unused	number of unused resources
atomic_t	ns_locks	number of locks in this namespace
__u64	ns_resources	number of resources in this namespace
ldlm_res_policy	ns_policy	callback function when you have a lock request with intent

Figure 7: The fields of *ldlm_namespace* data structure.

Some additional notes on `ns_client`: Each client only needs access to some portions of a namespace, but not all of them. So each client carries a so-called *shadow namespace*. `ns_client` is a flag that says this namespace is for client only and is not complete (see Figure 8 for an illustration). `ns_hash` is a hash table for all resources in this namespace. Notice that this is a pointer to the linked list, not just the linked list itself.

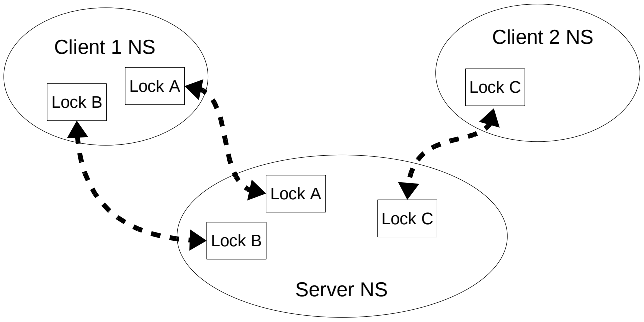

Figure 8: Shadow namespaces on clients.

4.2 Resource

A lock is for protecting resources. In Lustre, the most common type of resources are files. A **local lock** refers to a lock that is private, known only to the local entity. Correspondingly, a **global lock** is visible to the others. Note that "global" in this case may be a misnomer – it doesn't mean the lock is actually broadcast and known by all parties. In Lustre, it means you have a client copy of the lock, but server also got a copy through client request.[10]

[10] The lock on the server may be called the master lock.

A resource is defined in struct `ldml_resource`. Some highlights of the struct are discussed below.

`struct ldlm_namespace *lr_namespace` Points back to the namespace this resource belongs to. If the resource is a object id (oid), then its namespace is the associated OST. If it is fid, then its namespace is defined by the MDS.

`struct list_head lr_granted, lr_converting, lr_waiting` There are three lists defined here to categorize different locks and their states requested on this resource. *Granted list* is for all locks that have been granted use on this resource. *Waiting list* is for locks requesting the resource, but must wait due to a conflict. *Converting list* is for locks changing mode.

`struct ldlm_res_id lr_name` The name of the resource is defined in a struct, which is a 4×64 bits long identifier. All types of resources, either fid or oid, will be converted to this format as their generic resource name.

4.3 Lock Type and Mode

A lock has both a type and a mode. We discuss the mode first. There are six lock modes along with a compatibility matrix to indicate if two locks are compatible. The six modes are given below.

- `EX` **Exclusive mode** Before a new file is created, MDS requests `EX` lock on the parent.
- `PW` **Protective Write (normal write) mode** When a client requests a write lock from an OST, a lock with PW mode will be issued.
- `PR` **Protective Read (normal read) mode** When a client requests a read from an OST, a lock with `PR` mode will be issued. Also, if the client opens a file for execution, it is granted a lock with `PR` mode.
- `CW` **Concurrent Write mode** The type of lock that the MDS grants if a client requests a write lock during a file open operation.
- `CR` **Concurrent Read mode** When a client performs a path lookup, MDS grants `inodebit` lock with the `CR` mode on the intermediate path component.
- `NL` **Null mode**.

The corresponding matrix is given in Figure 9.

In Lustre, four types of locks are defined, and it is up to the client to decide what type of lock it requests. The component that requests a lock from the lock manager is the client, and it can be Lustre Lite, OSC or MDC. These four types of locks are given below.

extent lock for protecting OST data, defined by struct `ldlm_extent`.

flock for supporting a user space request on flock, defined by struct `ldlm_flock`.

	NL	CR	CW	PR	PW	EX
NL	1	1	1	1	1	1
CR	1	1	1	1	1	0
CW	1	1	1	0	0	0
PR	1	1	0	1	0	0
PW	1	1	0	0	0	0
EX	1	0	0	0	0	0

Figure 9: Lock compatibility matrix.

inode bit lock for protecting metadata attributes, defined by struct `ldlm_inodebits`.

plain lock defined, but not in use and can be ignored.

4.4 Callbacks

Another concept associated with the lock request is the *callback function*. When a lock is created, a client can supply three type of callbacks:

blocking callback There are two conditions where this callback will be invoked. First, if a client requests a lock conflicting with this one (even if the request comes from the same client), then this callback is invoked, so that if this client plays "nice" and has no use for this lock anymore, it can release the lock and let the other party have it. The second case is when a lock is revoked (after all references went away and the lock was cancelled).

completion callback There are also two cases in which this callback will be invoked: first, if the lock requested is granted; second, if the lock is converted, for example, to a different mode.

glimpse callback This callback is used to provide certain information about underlying properties without actually releasing the lock. For example, an OST can provide such a callback to provide information on file size since only the OSTs know exactly the size of a file object. This callback is then associated with the extent lock on the file object and is invoked by the server upon receiving the client request.

4.5 Intent

An intent is a piece of data supplied with a lock enqueue indicating special processing is needed during lock enqueue, and the data themselves are the parameters for that processing. Namespaces can have different *intent handlers* defined for such processing.

The intent feature allows the caller of namei and lookup to pass some information about what they want to accomplish in the end. The net effect of this is a reduced number of RPC requests when interacting with the MDS.

A intent is defined as follows (inode.h):

```
struct intent {
    unsigned    int_opmask;
    void        * int_arg1;
    void        * int_arg2;
};
```

Six intention operations are defined (encoded in int_opmask). These are get attributes, set attributes, insert or delete, open, create, and read links.

4.6 Lock Manager

Now let's provide a general description of how the locking algorithm (implemented by the lock manager) works in Lustre. The entry call to Lustre Distributed Lock Manager (LDLM) changes with different Lustre versions; we take 1.6 branch code as our reference. There are two major types of requests an LDLM can service: first, when a lock is requested, and second, when a lock is cancelled. We describe the two cases separately.

4.6.1 Requesting a Lock

1. A locking service client, whether it is a Lustre client, MDS, or OST, usually starts with a call to ldlm_cli_enqueue(). This function first examines if the lock requested belongs to the local namespace by checking the flag ns_client in the namespace struct. Remember that each LDLM instance defines its own namespace. If it is local, meaning that we don't have to send RPC to communicate, we skip to step 7; otherwise, we continue processing the lock remotely.

2. If the lock request is for a non-local namespace, we need to send *lock enqueue RPC* to LDLM running on the server; this is done inside the ldlm_cli_enqueue(). On the server side, ldlm_handle_enqueue() unpacks the lock request first, then creates a lock (ldlm_lock_create()). This lock is just an initial one with some fields filled in from the original lock request; it has not been granted yet. You can determine if a lock is granted or not by checking:

   ```
   if (lock->req_mode == lock->granted_mode) {   /* granted */
       ...
   }
   ```

   Now we move to the next step to check if the lock can be granted and what lock should be granted.

3. ldlm_lock_enqueue() is the core step for granting the lock. It checks if the lock intent is set. If no intent is set, then it checks for the lock type and invokes the policy function defined for this lock type. The policy function will determine if the request lock can be granted or not.

   If the lock intent is set, go to step 6.

4. For the resource for which the lock is being requested, two conditions need to be checked as given below.

    - Are there any conflicts with those granted locks?
    - Are there any conflicts with those waiting locks?

    If the answer to both questions is no, then no conflict is found, and the lock can be granted. Call *completion AST*[11], return the granted lock, and we are done. Otherwise, continue.

5. For each conflicting lock, we need to call *blocking AST*. Blocking AST checks if the lock is held at the server, then just sets a flag. If the lock is held at the client, then a *blocking RPC* request must be sent. Either way, after all these are done, put this new lock request on the waiting list, then return the lock with status as being blocked.

    A more detailed policy function implementation is provided later.

6. If the lock intent is set, then all that needs to be done is to invoke the intent handler. The key point is that LDLM does not interpret intents. The parties communicating with each other with intent do that; for example, MDS needs to register its intent handler, and OST needs to register its intent handler. In general, intent handler is registered per namespace by calling `ldlm_register_intent()`. It is the LDLM's responsibility to invoke them. The intent handler will determine if the lock can be granted or not. LDLM just returns the result of their verdict.

7. For a local lock request, it goes to a different branch at `ldlm_cli_enqueue_local()`, except that in this case it doesn't need to send an RPC anymore. It will need to go through the two phases described above: creating a lock first, then calling `ldlm_lock_enqueue()` to check if this can be granted. If either lock is granted or if there are any errors, it returns immediately with the lock marked correspondingly. Otherwise, lock request is being blocked, and it needs to wait for it.

    Please note that a server (e.g., MDS or OST) can initiate a lock request by directly calling `ldlm_cli_enqueue_local()` since they know the lock must be held by the local LDLM server.

4.6.2 Canceling a Lock

A lock is usually released involuntarily, and the owner will hold it as long as possible until: someone is asking for a conflicting lock, LDLM sends out a blocking AST, and the block AST handler invoked on the LDLM client side. Now we enter the canceling process. There are three counters in the lock that are related to the cancelling process. These can be listed as (1) counts of active readers, (2) counts of active writers, and (3) counts of users.

[11] AST is the acronym for Asynchronous System Trap from VMS lock design. Here, we consider it to be synonymous with a callback.

1. The entry point for cancelling a lock is `ldlm_cli_cancel()`. This function first checks if the sum of active readers and writers is zero. If not, it means another process on the same client is using the lock so, we do nothing. Eventually, the other client(s) will release the lock and follow the same code path and reach this checkpoint, then it will move to the next step.

2. Now when the sum of readers and writes count reaches zero, we call blocking AST with a flag set indicating this lock is being revoked.

3. Check if this lock is in the local namespace. If it is not, send *cancel RPC* to client (Lustre client). If so, the server side calls `ldlm_handle_cancel()` to cancel the lock.

   A cancel operation essentially involves steps that take this lock of all the lists it is currently on, such as granted list, waiting list, etc. Later, `obd_cancel()` will be called not to cancel the lock, but just to release the references on reader and writer counts.

4. Now that the lock has been cancelled, server can reprocess all waiting locks on the resource. It essentially goes through the same logic to determine if their lock requests can be granted.

5. If a waiting lock request can be granted, move it to the granted lock list, then invoke the *completion AST*.

4.6.3 Policy Function

As we discuss in the section on requesting a lock (4.6.1), policy function is being called to determine if the requested lock is in conflict with existing requests, based on the lock type. We have four lock types, and these are *inodebits*, *extent*, *flock*, and *plain*. Therefore, we need four policy functions. They all follow a similar overall flow, with some minor changes. In this section, we give an overall description on the overall flow.

1. There are two given parameters and these are lock request and a flag of `first_enq`. This flag means to signal if this is first time we do an enqueue request. If this is the first time, the waiting list needs to be processed and send blocking ASTs when necessary. If this is not the first time, then we already sent out blocking AST before, so there is no need to do it again.

   Another important thing about `first_enq` parameter is that when it is not set (which means we are reprocessing a lock already in the waiting list), we stop processing the waiting list after we find our own earlier entry in the list. Since the rest of the locks were enqueued at a later point in time, those can be ignored safely.

2. The policy function calls on a helper function for real work, passing in as parameters both lock list, enqueue flag, as well as a RPC list, either `NULL` or a empty

list. The first time, the helper function is called with a granted lock list. For each lock in the list, it performs the following conflict detection.[12]

(a) It always starts with mode checking. If two locks are compatible (refer to the previous section for the compatibility matrix), then no more checking is needed, and a lock can be granted. We continue only if locks are incompatible and the lock type is not **plain** since plain lock is not actually in use.

(b) For the **extent** lock type, if the interval or range does not intersect, then there is no conflict. However, within the boundary of a stripe size, LDLM always tries to grant the largest extent possible to reduce the possibility of further requests in case the client asks for more locks.

(c) For the **inodebits** lock type, if the bits requested (Lustre has a 64-bit lock space, but only 3 are being used) intersect, then there are conflicts. [13]

If the RPC list is `NULL`, we stop and return right after the first conflict is detected, since by passing `NULL`, we obtained all the information that the caller requires.

If the RPC list is not `NULL`, we need to continue for each remaining locks and add the conflicting lock into the RPC list.

3. If no conflicts are found, then the lock is granted. Otherwise, for each conflict lock in the RPC list, we invoke blocking AST.

4. If first enqueue flag is set, then we go back to step 2, but this time, we invoke the helper function with the lock list set as waiting list instead of granted list.

4.7 Use Cases

In this section, we use examples to give high level overview on processing lock requests.

MDS: One Client Read

Let's assume client C_1 wants to open the file `/d1/d2/foo.txt` to read. During the VFS path lookup, Lustre specific lookup routine will be invoked (see the Lustre Lite section for details on this). The first RPC request is lock enqueue with lookup intent. This is sent to MDS for lock on `d1`. The second RPC request is also lock enqueue with lookup intent and is sent to MDS asking inodebits lock for `d2`. The lock returned is an inodebits lock, and its resources would be represented by the fid of `d1` and `d2`.

The subtle point to note is, when we request a lock, we generally need a resource id for the lock we are requesting. However in this case, since we don't know the resource id for `d1`, we actually request a lock on its parent `/`, not on the `d1` itself. In the intent,

[12] An implementation can choose to implement its own policy function and conflict detection, not necessarily follow exactly as we presented here.

[13] It groups locks based on the bits (more precisely, it is based on the integers formed by the 3-bit value since only 3-bits are in use), so you only need to check one member in each group to make a decision, as a performance optimization.

we specify it as a lookup intent and the name of the lookup is d1. Then, when the lock is returned, the lock is for d1. This lock is (or can be) different from what the client requested, and the client notices this difference and replaces the old lock requested with the new one returned.

The third RPC request is a lock_enqueue with open intent, but it is not asking for lock on foo.txt.[14] That is, you can open and read a file without a lock from MDS since the content is provided by OST. Requesting locks from an OST is a different matter and is discussed later.

In other words, what happens at open is that we send a lock_request, which means we *do* ask for a lock from LDLM server. But, in the intent data itself, we might (or not) set a special flag if we are actually interested in receiving the lock back. And the intent handler then decides (based on this flag), whether or not to return the lock.

If foo.txt exists previously, then its fid, inode content (as in owner, group, mode, ctime, atime, mtime, nlink, etc.) and striping information are returned.

If client C_1 opens the file with the O_CREAT flag and the file doesn't exist, the third RPC request will be sent with open and create intent, but still there will be no lock requests. Now on the MDS side, to create a file foo.txt under d2, MDS will request through LDLM for another EX lock on the parent directory. Note that this is a conflicting lock request with the previous CR lock on d2. Under normal circumstances, a fourth RPC request (blocking AST) will go to client C_1 or anyone else who may have the conflicting locks, informing the client that someone is requesting a conflicting lock and requesting a lock cancellation. MDS waits until it gets a cancel RPC from client. Only then does the MDS get the EX lock it was asking for earlier and can proceed.[15]

If client C_1 opens the file with LOV_DELAY flag, MDS creates the file as usual, but there is no striping and no objects are allocated. User will issue an ioctl call and set the stripe information, then the MDS will fill in the EA structure.

MDS: Two Clients

If C_1 wants to create a file foo.txt under the directory /d1/d2, then no locks are required on foo.txt, but it will request a lock on d2 with the intent of instructing MDS to "open and create" the file foo.txt and possibly to return a lock on that file.

It acquires a CW lock intentionally on foo.txt. Now C_2 comes in and wants to create a new directory d3 under /d1/d2. At this point nothing will change as far as the CW lock is concerned.

OST: Two Clients Read and Write

After the MDS fills in the EA structure, the client has the file handle and stripe information. The client can now talk to OSTs. Let's assume we have four OSTs and two clients. Client C_1 reads file A and the second client C_2 writes to file A. We further assume that C_1 wants to read data objects, A_1, A_2, A_3, and A_4.

[14] Lustre Lite always decides no need for a lock unless it is from NFS.

[15] In the current code, client gets smart. When it sees that it will create a new file, it cancels the lock on d2 itself by embedding the cancel operation on the third RPC request. This saves two RPCs, one blocking AST from MDS to client and one cancel RPC from client to MDS.

First, client C_1 sends lock enqueue requests to OSTs 1 through 4 in parallel, asking for read lock with intent flag set. What it means is, if any of the objects client C_1 is asking has a blocking request, don't try to get the lock, just return back the information (file size, modification time, etc.) described by a data structure called `lvb` (Lustre Value Block) (the lock request is a glimpse because it has intent flag set).

If there are no conflicts, the client will be granted read locks on the entire file objects, with lock mode as `PR`. Now the client has four read locks on four OST file objects. Let's further assume that the client only wants to read A_1. Since it has the lock, it can now go ahead and read it.

Assume C_2 is writing to OST3. Instead of returning a `PR` lock, OST3 will contact C_2 and ask for the `lvb` information. It then would return this format back to C_1: "I am not issuing you the lock on A_3, but here is the state of that object".

This was done just to determine the file size, and we need to know the file size, for example, if we are trying to read beyond the file size. If the content of read falls into the object you already hold a `PR` lock on, then no further RPCs for that lock are needed.

Now if the content the client needs to read is on A_3, then it has no choice but to send a read lock request (without intent this time) to OST3. Within the request itself, the client should also make clear to what *extent* it is requesting the lock. If the extent it is requesting is not conflicting (intersecting reads) with the `PW` write lock C_2 is holding, then the read lock request will be immediately granted.

If a conflict exists, then OST3 will reply to C_1 that the lock request is not granted and at the same time, it will send a blocking request to C_2. There are two cases to consider from the perspective of C_2:

1. If the `write()` system call is done, meaning that all data has been written to some buffer (maybe not to disk yet), then the buffers will have the dirty flag set. In that case, the client flushes all dirty buffers, probably through one or more bulk writes. Then it releases the lock (lock reference count is now zero) by sending a cancel RPC to OST3.

2. If the `write()` is still in progress, then the system call hasn't returned yet. At this point, the lock reference count is non-zero and the blocking AST won't reach the lock logic. Only after the write system call finishes (it actually means at least two things here, one, the data is all in cache and two, the lock reference count is decreased to zero), then it can go to the step explained above.

After OST3 gets the cancel request from C_2, it will send a completion AST to client C_1, telling that now its lock request is granted.

We don't explain a case where both clients do read because it is trivial, as both of them can get `PR` locks, even with their extent intersecting.

If one client doesn't play nice and doesn't cooperate in releasing its lock, then a timer is started the moment it was sent a blocking AST. If the client doesn't release the lock and there is no ongoing I/O under that lock for the duration of the timeout, then the client is evicted. However, if there is ongoing I/O, then on every I/O RPC the timeout is prolonged.

5 OST and obdfilter

```
/* Sigh - really, this is an OSS, the _server_, not the _target_ */
static int ost_setup(struct obd_device *obd, obd_count len, void *buf)
{ ... }
```

from Lustre source tree b16

Lustre source tree `lustre/ost` and all function names prefixed with `ost_` should probably be regarded as server (OSS) functions, if we understand the above comment correctly.

5.1 OSS as OST

OST is loaded as a kernel module. It works closely with obdfilter and does most of the server/OST side of the work. Between these two layers, OSS is the switch layer or the *thin* layer, and it interprets requests from Portal RPC, prepares for requests, and then passes requests to obdfilter for further processing. In the following discussion, we focus on two aspects of it: initial setup and switching structure, implemented by `ost_setup()` and `ost_handle()`, respectively.

Initial Setup

- First, the OST checks if the OSS thread number is specified. If not, then it computes the minimum number of threads based upon the CPU and memory and ensures that there is 4x dynamic range between the minimum and maximum number of threads.

  ```
  oss_min_threads = num_possible_cpus() * num_physpages >> (27 - CFS_PAGE_SHIFT);
  if (oss_min_threads < OSS_THREADS_MIN)
       oss_min_threads = OSS_THREADS_MIN;
  /* Insure a 4x range for dynamic threads */
  if (oss_min_threads > OSS_THREADS_MAX / 4)
       oss_min_threads = OSS_THREADS_MAX / 4;
  oss_max_threads = min(OSS_THREADS_MAX, oss_min_threads * 4 + 1);
  ```

  To get the obd device of the OST; the following function call is used.

  ```
  struct ost_obd *ost = &obd->u.ost;
  ```

- Then, the server side initiates RPC services by:

  ```
  ost->ost_service = ptlrpc_init_svc( , , , , , , ost_handle, , , , "ll_ost");
  ```

  The function returns the pointer to struct `ptlrpc_service`. One important thing to note is that we have supplied a handler, `ost_handle`. Once the service is set up as shown below, Portal RPC will dispatch the request to this handler for further processing. That is the subject of the following section.

- The prtrpc threads are started as:

  ```
  rc = ptlrpc_start_threads(obd, ost->ost_service);
  ```

- The similar call sequence is repeated for creating *ost create* threads and the returned service handle is assigned to `ost->ost_create_service`. It is also repeated for *ost io* threads, and a service handle is assigned to `ost->ost_io_service`.
- And finally, the ping eviction service is started.

Dispatching

The handler function takes one input parameter, `struct ptlrpc_request *req`, and it is driven largely by the type of request. The decoding of type of request is through passing `req->rq_reqmsg` (which points to struct `lustre_msg`) to a helper function `lustre_msg_get_opc()` provided by Portal RPC. Thus the dispatch structure looks like:

```
swtich (lsutre_msg_get_opc(req->rq_reqmsg)) {
case OST_CONNECT:
    ...
    rc = target_handle_connect(req, ost_handle);
    break;
case OST_CREATE:
    ...
    rc = ost_create(req->rq_export, req, oti);
    break;
case OST_WRITE:
    ...
    rc = ost_brw_write(req, oti);
    RETURN (rc);
case OST_READ:
    ...
    rc = ost_brw_read(req, oti);
    RETURN(rc);
}
```

The exception handling includes the possible recovery, which can happen during any request except the `OST_CONNECT`. Also, we need to check for connection coming from an unknown client by checking `NULL` of `req->rq_export`.

5.2 OST Directory Layout

This section describes what you will observe on the disk when logging onto an OST node. The filesystem on the disk is most likely **ldiskfs** for now. It means the backend data is really stored as a regular file, organized in a certain Lustre specific way:

Group Number

Under the top level directory on an OST is the subdirectory named for each group. This layout accommodates clustered MDSs where each group corresponds to one MDS. As of now, only one MDS is in use, so only group zero is effective. [16]

[16] In fact, there is a special group for echo client as well, so that MDS and echo client do not conflict when run at the same time.

Object Id

Under each group, 32 subdirectories are created. For each file object, its last five digits are used to identify which subdirectory this file should reside with. Here, the filename is the object id.

5.3 obdfilter

The obdfilter device is created when the OST server is initialized. For each OST, we have an associated obdfilter device. For each client connection, the obdfilter creates an export as the conduit of communication. All the exports are maintained in a global hash table, and the hash key is also known as UUID, as shown in both Figure 10 and 11. The Portal RPC layer makes use of UUIDs to quickly identify to which export (and obdfilter device) the incoming request should go. Also, each obdfilter device maintains a list of the exports it is serving. This relationship is illustrated in Figure 10.

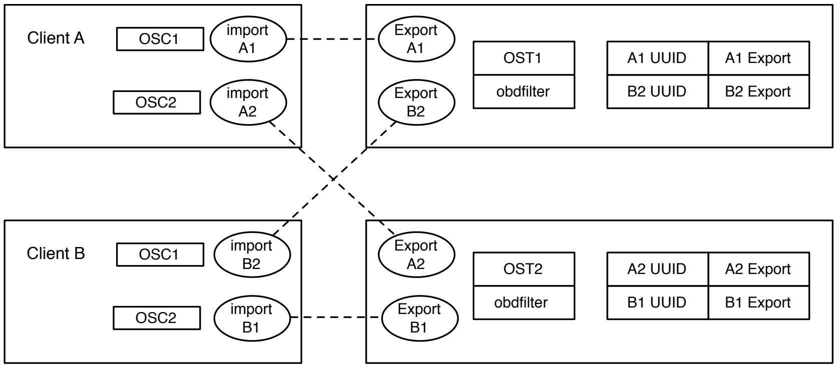

Figure 10: Import and export connections between an OSC and obdfilter.

The obdfilter provides the following functions:

- handles create request, presumably from MDS for file data objects.
- handles read and write requests, from OSC clients.
- handles connect and disconnect requests from lower Portal RPC layer for established exports and imports.
- handles destroy (which involves both client and MDS) requests.

5.3.1 File Deletion

The destroy protocol is as follows. First, the client decides to remove a file and this request is sent to MDS. MDS checks the EA striping and uses `llog` to make a transaction log. This log contains the following: *<unlink object 1 from ost1, unlink object*

2 from ost2, etc.>. Then, MDS sends this layout and transaction log back to the client. The client takes this log and contacts each OST (actually obdfilter) to unlink each file object. Each successful removal is accompanied by a confirmation or acknowledgment to the unlink `llog` record. Once all unlink llog records on MDS have been acknowledged, the file removal process is completed.

5.3.2 File Creation

As discussed earlier in Section 5, all requests are handled by OST and obdfilter together. Now, we will walk through the handling of a create request. The first portion of request handling is done inside `ost_create()` as follows.

1. Prepare the size of reply message. It is composed of two records and thus requires two buffers. The first record is for portal rpc body, and the second, for the ost reply body.

   ```
   __u32 size[2] = { sizeof(struct ptlrpc_body), sizeof(*repbody)};
   ```

2. Get a pointer to the request body from the original request and do byte swapping when necessary.

   ```
   struct ost_body *body = lustre_swab_reqbuf(req, REQ_REC_OFF,
                 sizeof(*body), lustre_swab_ost_body);
   ```

   The last parameter is the swab handler, which is called only when the swapping is necessary. Client side uses native byte order for its request, along with a pre-agreed magic number. Server side reads the magic number to decide if a swap is needed.

3. Do the actual space allocation, and fill in preliminary header information.

   ```
   rc = lustre_pack_replay(req, 2, size, NULL);
   repbody = lustre_msg_buf(req->rq_repmsg, REPLY_REC_OFF,sizeof(*repbody));
   ```

   After the first call, `req->rq_repmsg` points to the newly allocated space. The second call assigns `repbody` of the starting address for the buffer of the reply body.

4. Finally, it fills in the reply body with exactly the same contents as a request body and passes on to obdfilter for further processing.

   ```
   memcpy(&repbody->oa, &body->oa, sizeof(body->oa));
   req-rq_status = obd_create(exp, &repbody->oa, NULL, oti);
   ```

For the create request, the entry point for obdfilter is through `filter_create()`.

```
static int filter_create(struct obd_export *exp, struct obdo *oa ..)
```

We ignore the processing related to `struct lov_stripe_md **ea` and `struct obd_trans_info *oti` because the former is a legacy code and is unlikely to be used in the future.

1. First, save the current context and assign this client its own operation context. This is for specifying necessary information for the thread if it wants to access the backend filesystem. It is like a sandbox limiting the reach of server threads when processing requests from clients; it stores “filesystem root” and “current working dir” for the server thread (not obtained from the client, of course, but rather dependent on which OST we are working on).

```
obd = exp->exp_obd;
put_ctxt(&saved, &obd->obd_lvfs_ctxt, NULL);
```

2. If the request is for recreating an object, then we cancel all extent locks on the recreated object by acquiring the lock on the object and call on `filter_recreate()` to do the actual job. Otherwise, we follow the normal flow of precreating objects. The reason for precreating is that, conceptually, when MDS asks an OST for creating an object, OST doesn’t just create one object, it creates multiple objects with object id assigned. These batch created objects have a disk size of zero. The goal is, when MDS responds to a client request next time for creating new file, it doesn’t have to send a request to OST again to present the layout information to client. By taking a look at the pool of precreated objects from each OST, MDS may already have all the information needed to reply to the client.

```
if (oa->o_valid & OBD_MD_FLFLAGS) &&
        (oa->o_flags & OBD_FL_RECREATE_OBJS)) {
    rc = ldlm_cli_enqueue_local(obd->obd_namespace, &res_id, ... );
    rc = filter_recreate(obd, oa);
    ldlm_lock_deref(&lockh, LCK_PW);
} else {
    rc = filter_handle_precreate(exp, oa, oa->o_gr, oti);
}
```

   Here, `rc` returned from precreate handler is either a negative, indicative of an error, or a non-negative number, representing the number of files created.

3. Now, we take a closer look at the function of precreate:

   When a client contacts an OST with a precreated object id, OST knows that this object id now is activated. However, this presents a problem such that, if the MDS has failed, it now has stale information on precreated objects. To resolve this conflict, when MDS is restarted, it checks its records on unused precreated objects and sends requests to OSTs to delete these objects (*delete orphans*). The obdfilter takes these requests and skips those objects that are actually in use (but out of synchronization with MDS’s own record) and removes the rest of it. This is the first part of what `filter_handle_precreate()` will do:

```
if ((oa->o_valid & OBD_MD_FLFLAGS) &&
    (oa->o_flags & OBD_FL_DELORPHAN)) {
    down(&filter->fo_create_lock);
    rc = filter_destroy_precreated(exp,oa,filter);
    ...
} else {
    rc = filter_precreate(obd, oa, group, &diff);
    ...
}
```

4. Finally, the create request is passed onto `fsfilt` and is completed by a VFS call. The process will later go through more steps, such as getting hold of parent inode, transaction setup, etc.

```
rc = ll_vfs_create(dparent->d_inode, dchild,
                    S_IFREG | S_ISUID | S_ISGID | 0666, NULL);
```

6 MDC and Lustre Metadata

6.1 MDC Overview

The MDC component is layered below Lustre Lite. It defines a set of metadata related functions that Lustre Lite can call to send metadata requests to MDS. The set of functions is implemented in `lustre/mdc`, and we discuss them in Section 6.3.

Lustre Lite passes the request parameters in a `mdc_op_data` data structure, and all requests must eventually be converted to a structure known as the `ptl_request`. Therefore, before a RPC request can be carried out, there are several steps for preparing it (*packing*). Within the `mdc_lib.c`, there are a number of functions defined for this purpose. In turn, some of these functions actually invoke packing helper methods provided by the Portal RPC layer.

Once `ptl_request` is ready, MDC can then invoke `ptlrpc_queue_wait()` to send the request. This is a synchronous operation, and all metadata operations with intent are also synchronous. There are other ptlrpc methods for send operations and used for metadata operations without intent (*mdc_reint*)—this is done in code `mdc_reint.c`.

For the most part, the driver of converting from `mdc_op_data` to `ptl_request` and the call to enqueue the request are done in function `mdc_enqueue`, implemented in `lustre/mdc/mdc_locks.c`.

6.2 Striping EA

Depending on where the striping EA is created or used, there are three formats:

- On-disk format, used when it is stored on MDS disk, described by `struct lov_mds_md`.
- In-memory format, used when it is read into the memory and unpacked, described by `struct lov_stripe_md`.
- User format, used when the information is to be presented to the user, described by `struct lov_user_md`.

The user format differs from on-disk format in two aspects:

- The user format has `lmm_stripe_offset`, which on-disk format does not have. This field is used to transfer the `striping_index` parameter to Lustre when user wants to set a stripe.
- The user format has `lmm_stripe_count` as 16-bit, whereas on-disk format has this field as 32-bit.

6.3 Striping API

Five types of APIs are defined to handle striping EA. These are listed below.

set/get API To set or get a stripe EA from storage. It operates on on-disk striping EA.

```
int fsfilt_set_md(struct obd_device *obd, struct inode *inode,
               void *handle, void* md, int size, const char *name)
int fsfilt_get_md(struct obd_device *obd, struct inode *inode,
               void *md, int size, const char *name)
```

Here, `md` is the buffer for the striping EA, `handle` is the journal handle, and `inode` refers to the MDS object.

pack/unpack API As striping EAs are stored in packed format on disk, they are unpacked after `fsfilt_get_md()` is called. These APIs can be used for both on-disk and in-memory striping EA.

```
int obd_packmd(struct obd_export *exp, struct lov_mds_md **disk_tgt,
               struct lov_stripe_md *mem_src)
int obd_unpackmd(struct obd_export *exp, struct lov_stripe_md **mem_tgt,
                struct lov_mds_md *disk_src, int disk_len)
```

Here, `mem_src` points to the in-memory structure of striping EA, and `disk_tgt` points to the disk structure of striping EA. Conversely, `disk_src` is the source for on-disk striping EA, while `mem_tgt` is the target for in-memory striping EA.

alloc/free API Allocates and frees striping EA in memory and on-disk.

```
void obd_size_diskmd(struct obd_export *exp, struct lov_mds_md *dis_tgt)
int obd_alloc_diskmd(struct obd_export *exp, struct lov_mds_md **disk_tgt)
int obd_free_diskmd(struct obd_export *exp, struct lov_mds_md **disk_tgt)
int obd_alloc_memmd(struct obd_export *exp, struct lov_stripe_md **mem_tgt)
int obd_free_memmd(struct obd_export *exp, struct lov_stripe_md **mem_tgt)
```

striping location API Returns data object location information from striping EA.

```
obd_size love_stripe_size(struct lov_stripe_md *lsm, obd_size ost_size, int stripeno)
int lov_stripe_offset(struct lov_stripe_md *lsm, obd_off lov_off,
                   int stripeno, obd_off *obd_off)
int lov_stripe_number(struct lov_stripe_md *lsm, obd_off lov_off)
```

Here, `lov_off` is the file logical offset. The `stripeno` is the stripe number of the data object.

lfs API User-level API to handle striping EA and is used by the `lfs` utility.

```
 int llapi_file_get_stripe(const char *path, struct lov_user_md *lum)
 int llapi_file_open(const char *name, int flags, int mode,
    unsigned long stripe_size, int stripe_offset, int stripe_count,int stripe_pattern)
```

Here, `llapi_file_get_stripe()` will return a user striping EA given a path, and `llapi_file_open()` opens/creates a file with a user specified stripe pattern. It is also worth noting that `stripe_offset` is the same as `stripe_index` used in user space, and it is the OST index of the first stripe.

7 Infrastructure Support

This section discusses miscellaneous aspects related to Lustre initialization, client registration, obd devices management, etc.

7.1 Lustre Client Registration

Lustre client side or Lustre Lite registers as a filesystem with the name "lustre." The file system type is defined as:

```
    struct file_system_type lustre_fs_type = {
        .owner      = THIS_MODULE,
        .name       = "lustre",
        .get_sb     = lustre_get_sb,
        .kill_sb    = lustre_kill_super,
        .fs_flags   = FS_BINRARY_MOUNTDATA | FS_REQUIRES_DEV
                        LL_RENAME_DOES_D_MOVE,
    };
```

Another function defined by `lustre_register_fs()` simply invokes kernel function:

```
    return register_filesystem(&lustre_fs_type);
```

and will be done with the registration. This was invoked when `obdclass` as a module was being initialized.

```
int init_obdclass(void) {
...
#ifdef __KERNEL__
    err = lustre_register_fs()
#endif
}
```

See `class_obd.c` for more details.

7.2 Superblock and Inode Registration

It should be a relatively simple matter, but Lustre jumps through several hoops that may be due to some legacy support issues.

When Lustre Lite initializes as a Linux module, `init_luster_lite()` was defined in `super25.c` [17] Among things such as allocating inode cache, it assigns (essentially) the function pointer `*client_fill_super` with `ll_fill_super`.

In the `luster_fill_super()` method, we check if this is a client mount. If it is, it makes calls to `(*client_fill_super)(sb)`, where `sb` is the superblock.

[17] It is actually used for Linux Kernel 2.6 as well.

7.3 OBD Device

OBD device is meant to provide a level of abstraction on Lustre components such that the generic operations can be applied without knowing the specific devices you are dealing with. We discussed the method tables defined on it in Section 5. Lustre also provides some generic infrastructure for managing object devices. The core structures are `obd_devices` and `exports`. We discuss them in detail.

Each obd device was assigned an integer number, there are at most `MAX_OBD_DEVICES` obd devices that can be created on a given node. You can retrieve an obd device through an integer number (`obd_minor`), a name, or a uuid. All obd devices are stored in an array internally. However, the preferred way to retrieve an obd device is through a set of API as discussed below.

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

The API can be roughly divided into five categories:

- Register and unregister a device type.
- Allocate and free obd devices. `obd_device_alloc()`, `obd_device_free()`.
- Create and release obd devices. You can create new obd devices by giving a string type and string name through `class_newdev()`. You can release one by giving a pointer to `obd_device` that it is to be released. In both cases, they internally invoke allocation and free functions.
- Search. You can search by type through `class_search_type()`;
- Conversion utilities.

7.4 Import and Export

For two OBD devices A and B to communicate, they need an import and export pair: A_{import} and B_{export}. A_{import} is for sending requests and receiving replies, and B_{export} is for receiving requests and sending replies. However, the same pair cannot be used the other way around: that is you cannot send a request through B_{export}. For that to happen, such as in the case of sending ASTs, you need so-called “reverse import.” Essentially, reverse import converts the A_{import} and B_{export} pair into an A_{export} and B_{import} pair. The need for establishing such a pair at OBD devices is partially illustrated in Figure 11.

Note that LOV, OSC, and OST in the figure are all defined as types of OBD devices.

8 Portal RPC

Portal RPC provides basic mechanisms for

- sending requests through imports and receiving replies,
- receiving and processing requests through exports and sending replies,
- performing bulk data transfer, and
- error recovery.

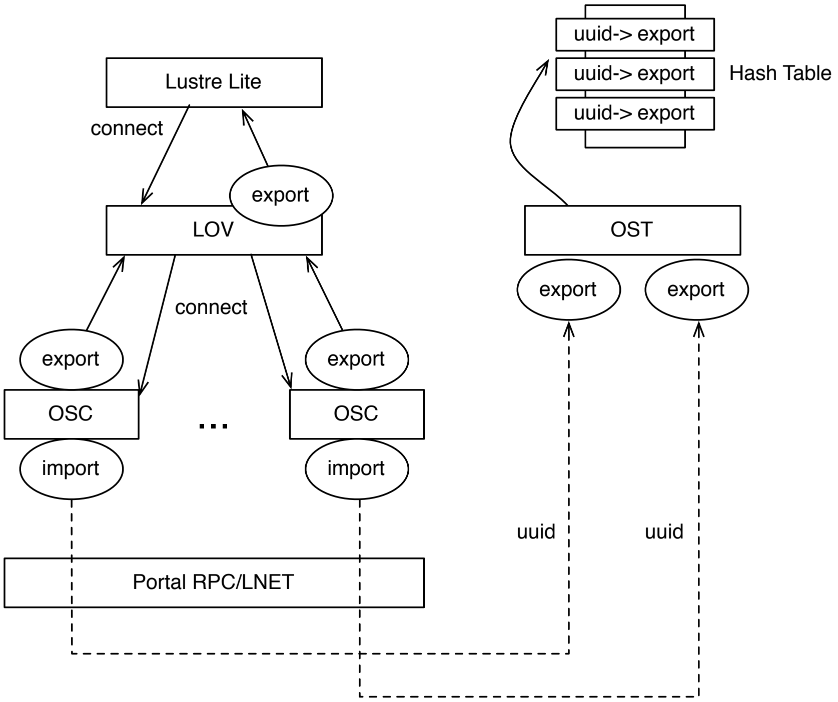

Figure 11: Setup export and import.

8.1 Client Side Interface

We will first discuss the interface of Portal RPC without going into implementation details. We will use the LDLM's send mechanism as an example. For this example, LDLM is sending a blocking AST RPC to a client that is the owner of a given lock (`ldlm_server_blocking_ast`) and also acting as the lock manager. This example will help us better illustrate how a client uses Portal RPC API.

First, we prepare the size for this request as given below.

```
struct ldlm_request *req;
__u32 size[] = { [MSG_PTLRPC_BODY_OFF] = sizeof(struct ptlrpc_body),
    [DLM_LOCKREQ_OFF] = sizeof (*body) };
```

A request can be seen as a sequence of records, where the first record has an offset of 0, the second record has an offset value 1, and so on. Once the size is determined, we can invoke `ptlrpc_prep_req()`. The prototype of this function is as follows:

```
struct ptlrpc_request *
  ptlrpc_prep_req(struct obd_import *imp, __u32 version, int opcode,
                int count, __u32 *lengths, char **bufs)
```

The RPC communication needs an import on the client's side, and it is created during the connect phase. The `*imp` points to this particular import, and `version` specifies the version to be used by the Portal RPC internals to pack the data. Packing here is the on-the-wire format that defines how the buffers are actually situated in the network packet. Each subsystem defines a version number to distinguish its layout request—for example, MDC and MDS define their version, MGC and MGS define another.

The `opcode` specifies what this request is about. Each subsystem defines a set of operation codes (see `lustre_idl.h` for more information). The `count` is the number of buffers needed for this request, and `*length` is an array with each element specifying the size of the corresponding requested buffer. The final parameter signals Portal RPC to accept (copy the data over) and process the incoming packet as is. For our example, this function is invoked as given below.

```
req = ptlrpc_prep_req(lock->l_export->exp_imp_reverse,
    LUSTRE_DLM_VERSION, LDLM_BL_CALLBACK, 2, size, NULL);
```

This declaration indicates that there are two buffers requested and the size of each buffer is represented by `size` in the parameter list.

In addition to the housekeeping, the above call allocates the request buffer and saves it in `req->rq_reqmsg`. The address of the interested record can be acquired by specifying the record offset:

```
body = lustre_msg_buf(req->rq_reqmsg, DLM_LOCKREQ_OFF, sizeof(*body));
```

On the server side, we see this same helper method with similar input parameters used to extract the field of interest. Once the buffer structure is obtained, necessary fields for the request can be further filled in. After all is done, there are several ways to send the request, as listed here.

`ptl_send_rpc()` A primitive form of RPC sending, which doesn't wait for a reply and doesn't try to resend when failure occurs. It is not a preferred way of doing RPC send.

`ptlrpcd_add_req()` A completely asynchronous RPC sending, handled by the ptlrpcd daemon.

`ptlrpc_set_wait()` A synchronous RPC sending of multiple messages, which returns only when it gets if all of the replies back. First, `ptlrpc_set_add_req()` must be used to add the request to a pre-initialized set, which contains one or more requests that must be sent together.

`ptlrpc_queue_wait()` Probably the most common way of sending RPC, which is synchronous and returns only after an RPC request is sent and a reply is received.

The last step after invoking this RPC request is to release the resource references by calling `ptlrpc_req_finished(req)`.

8.2 Server Side Interface

The server side uses Portal RPC in a completly different way from the client side. First, it initializes the service with the function call given below.

```
struct ptlrpc_service * ptlrpc_init_svc (
    int nbufs,              /* num of buffers to allocate */
    int bufsize,            /* size of above requested buffer */
    int max_req_size,       /* max request size server will accept */
    int max_reply_size,     /* max reply size server will send */
    int req_portal,         /* req service port in lnet */
    int rep_portal,         /* rep service port in lnet */
    int watchdog_factor,    /* wait time for handler to finish */
    svc_handler_t handler,  /* service handler function */
    char *name,             /* service name */
    cfs_proc_dir_entry_t *proc_entry, /* for procfs */
    svcreq_printfn_t svcreq_printfn,  /* for procfs */
    int min_threads,         /* min # of threads to start */
    int max_threads,         /* max # of threads to start */
    char *threadname         /* thread name prefix */
)
```

Once the call returns, the request can come in and the registered handler function will be called. Usually, server divides the task at hand into several types. For each type, it creates a different pool of threads. These threads could share the same handler. The reason for different pools is to prevent starvation. In some cases, multiple pools also prevent deadlocks, where all threads are waiting for some resource to become free to handle a new RPC.

8.3 Bulk Transfer

The client first sends a bulk RPC request. Let's assume this is a write request. It contains descriptions of what to send. Now the server processes the request, allocates the space, and then takes control of the data transfer. The next RPC from the server will perform a bulk transfer of the data to the pre-allocated spaces. One such example is done in the `osc_brw_pre_request()`. Let's walk through the process:

1. Bulk transfer is initiated with the preparation as discussed before. However, the prepare request is different in the sense that we are asking the request from a pre-allocated pool, which is the case if the request itself can be associated with a low-memory situation.

   ```
   req = ptlrpc_pre_req_pool(cli->cl_import, LUSTRE_OST_VERSION,
                                    opc, 4, size, null, pool)
   ```

   The `opc` in this case can be, for example, `OST_WRITE`.

2. Next, we specify the service portal. In the import structure, there is a default portal that this request will go, but in this case for the sake of the example, let's assume the request will be handled by a specific portal:

   ```
   req->rq_request_portal = OST_IO_PORTAL;
   ```

3. Then, we need to prepare the bulk request. We pass in as parameters the pointer to the request, number of pages, type, and destination portal. The return is a bulk descriptor for this request. Notice that the bulk request goes to a different portal:

```
struct ptlrpc_bulk_desc desc = ptlrpc_prep_bulk_imp(req, page_count,
             BULK_GET_SOURCE, OST_BULK_PORTAL);
```

4. For each page to be transferred, we invoke the `ptlrpc_prep_bulk_page()` and add one page at time to the bulk descriptor. There is a flag in the request indicating that this is a bulk request and we should check this descriptor to obtain all the layout information on pages.

```
struct ptlrpc_request {
    ...
    struct ptlrpc_bulk_desc *rq_bulk;
    ...
}
```

At the server side, the overall preparation structure is similar, but instead of preparing for import, now it prepares for an export. An example of it can be seen in `ost_brw_read()` in ost handler.

```
desc = ptlrpc_prep_bulk_exp(req, npages, BULK_PUT_SOURCE, OST_BULK_PORTAL);
```

The server side also needs to prepare each bulk page. Later, the server side can start the transfer by:

```
rc = ptlrpc_start_bulk_transfer(desc);
```

At this point in time, the first RPC request from the client has been processed by the server, and the server is ready for the bulk data transfer. Now the server can start the bulk RPC transfer as we mentioned at the begining of this section.

NRS Optimization

On the server side, another point to note is that we can receive a huge number of descriptors that describe the page layout to read or write. This presents an opportunity for an optimization if there are any neighboring reads or writes going to the same region. If there are, perhaps they can be grouped and processed together. That is the subject of the Network Request Scheduler (NRS). This also displays the significance of a two-phase bulk transfer, which allows us to get an idea of the incoming/outgoing data without actually getting the data first, so that they can be reorganized for better performance. The second reason for the two-phase operation is that as the service initialization increases on the server, a certain amount of buffer space is allocated. When client requests come in, they will be buffered in this space first before further processing, as pre-allocating a huge amount of space there just to accommodate potential bulk transfers is not preferred. Also, it is important not to overwhelm server buffer space with big data chunks, and two-phase operation helps in that context as well.

8.4 Error Recovery: A Client Perspective

Most of the recovery mechanism is implemented at the Portal RPC layer. We start with a portal PRC request which is passed down from the upper level. Inside Portal RPC, there are two lists maintained by the import that are important to our discussion.

These are the *sending* and *replay* lists. The import also maintains `imp->states` and `imp->flags`. The possible states are *full*, *connecting*, *disconnecting*, and *recovery*, and flags can be *invalid*, *active* and *inactive*.

After the health status of the import is checked, the send request will continue. The sequence of the steps is outlined here:

1. Send the RPC request, then save it into the sending list. Also start the **obd timer** at the client side.

2. If a server reply is received before the timeout expires and the request is *replayable*[18], then unlink it from the sending list and link it to the replay list. If the request is not replayable, then upon receiving the reply, remove the request from the sending list.

   A reply from the server doesn't necessarily mean it committed the data to the disk (assuming the request alters on-disk data). Therefore, we must wait for a transaction commit (a transaction num) from the server, which means that the change is now safely committed to the disk. This last server-committed transaction number is usually piggybacked with every server reply.

   Usually, a request from MDC to MDS is replayable, but an OSC to OST request is not, and this is only true if the asynchronous journal update is not enabled. There are two reasons:

   - First, a data request (read or write) from OSC to OST can be very large, and keeping them in memory for replay can be a huge burden on memory.
   - Second, OST uses only direct I/O (at least for now). The reply itself, along with transaction number, is enough of a guarantee to say the commit is done.

3. If a timer expires, then client marks this import state from *full* to *disconnect*. Now the pinger kicks in and if the server responds to the pinger, then the client will try to reconnect (connect with reconnect flags).

4. If the reconnect is successful, then we start the recovery process. We now mark the state as *recovery* and start sending the requests in the replay list first, followed by the requests in the sending list.

The key point about the pinger is that if requests are being sent frequently enough, then the pinger is not needed. It is activated only if a client has an extended idle period, and the pinger is used to keep the connection alive with the server so that it will not get evicted due to inactivity. On the other hand, if client went offline for whatever reason, the server will not get pinged by the client, and the server can still evict this client.

[18] Replay request only refers to those that will modify the on-disk data. For example, read is not a replayable request, but during recovery, they can still be resent if they are in the sending list.

9 LNET: Lustre Networking

LNET is a message passing API, originated from Sandia Portals. Although there are commonalities between the two, they are not the same thing. We will cover the Lustre LNET without delving into the differences between the two.

9.1 Core Concepts

First, we need to clarify some terminology used throughout the rest of the section, in particular, the process id, matching entry, matching bits, and memory descriptor.

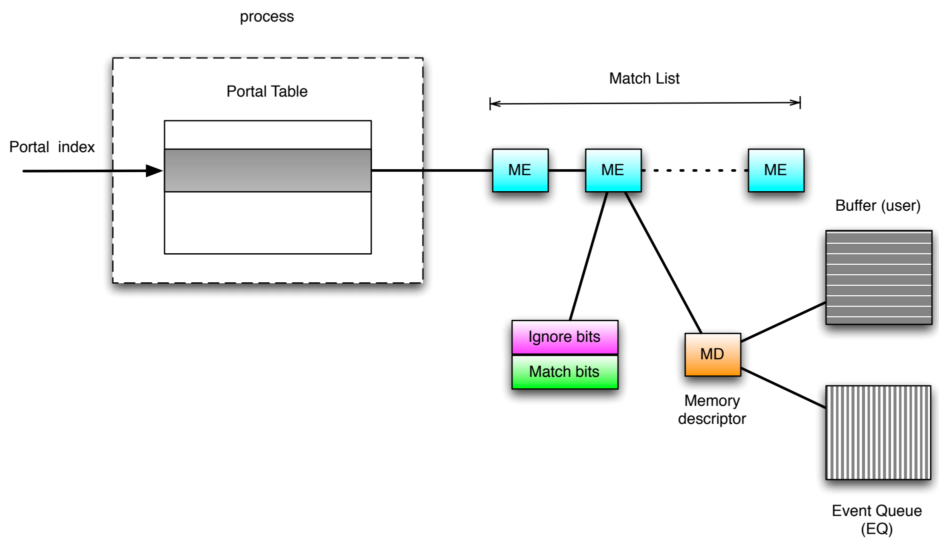

Figure 12: Illustration of Lustre LNET addressing scheme.

LNET Process Id

LNET identifies its peers using the LNET process id, defined as follows:

```
typedef struct {
    lnet_nid_t  nid;
    lnet_pid    pid;
} lnet_process_id_t;
```

The `nid` identifies the id of the node, and `pid` identifies the process on the node. For example, in the case of socket LND (and for all currently existing LNET LNDs), there is only one instance of LNET running in the kernel space; the process id therefore uses a reserved ID (12345) to identify itself.

ME: Matching Entry

A *portal* is composed of a list of match entries (ME). Each ME can be associated with a buffer, which is described by a memory descriptor (MD). ME itself defines match bits and ignore bits, which are 64-bit identifiers that are used to decide if the incoming message can use the associated buffer space. Figure 9.1 illustrates the Lustre LNET addressing scheme.

```
typedef struct lnet_me {
        struct list_head   me_list;
        lnet_libhandle_t   me_lh;
        lnet_process_id_t  me_match_id;
        unsigned int       me_portal;
        __u64              me_match_bits;
        __u64              me_ignore_bits;
        lnet_unlink_t      me_unlink;
        struct lnet_libmd *me_md;
} lnet_me_t;
```

All MEs associated with the portal are linked by the `me_list`. The `me_match_id` defines which remote LNET peer(s) are allowed to access this ME, and it can be a wildcard that allows open access.

MD: Memory Descriptor

Following the creation of an MD by the upper layer, the LNET layer uses the `struct lnet_md_t` to reference the MD. The LNET internal representation is described by `struct lnet_libmd_t`. Per our understanding, the purpose is to make `lnet_libmd_t` opaque to the client so that they cannot interfere with LNET internal states. They share some of the common fields, but LNET maintains more states for internal housekeeping.

```
typedef struct {
        void              *start;
        unsigned int       length;
        int                threshold;
        int                max_size;
        unsigned int       options;
        void              *user_ptr;
        lnet_handle_eq_t  eq_handle;
} lnet_md_t;
```

If a memory buffer described by MD is contiguous, then the address of `start` points to the begining of the memory. Otherwise, it points to the start of some I/O vectors. There can be two kinds of I/O vectors: if the memory is already mapped to virtual memory, it is described by `struct iovec`; otherwise, it is described by `lnet_kiov_t`, which may or may not be mapped to virtual memory, and by definition, it is just a memory page. MD options (`options`) identifies the type of I/O vectors. It is either `LNET_MD_IOVEC` or `LNET_MD_KIOV`. Also, if the MD is describing a non-contiguous memory, `length` then describes the number of entries in the array.

As mentioned above, `struct lnet_libmd_t` is used by LNET internal to describe the MD along with some other bookkeeping fields:

```
typedef struct lnet_libmd {
    struct list_head    md_list;
    lnet_libhandle_t    md_lh;
    lnet_me_t           *md_me;
    unsigned int        md_length;
    unsigned int        md_offset;
    unsigned int        md_niov;
    union {
        struct iovec    iov[LNET_MAX_IOV];
        lnet_kiov_t     kiov[LNET_MAX_IOV];
    } md_iov;
    ...
}
```

It should be obvious that md_me is the address of the ME entry this MD is associated with, and it can be NULL. The md_length is the total number of bytes of all segments this MD describes, and md_offset is the number of bytes to skip. For a noncontiguous memory, think of it as all being combined into a virtual contiguous array and you have a logical offset into it. The md_niov is the number of valid entries of the I/O vectors described by either iov or kiov in the union struct. md_list is a hashtable with handle (md_lh) as key to locate an MD.

Example Use of Offset

Upon initialization, the server will *post* request buffers (for the request portal); the buffer is to accommodate incoming client requests. We further assume that the request buffer is 4KB in size and each request should be 1KB at most. When the first message comes, the offset is increased to 1KB, and with the second message, the offset is set to 2KB and so on. So essentially, the offset is used to keep track of the write position to fend off an overwrite, and this is the default case. Another case where offset is being used differently will be described after we talk about MD options.

MD Options

If the MD has the LNET_MD_MANAGE_REMOTE flag set, then the client can indicate the offset into the MD for the GET or PUT operations. In the follow-on API discussion, we describe the offset parameter in the GET and PUT API. We describe two use cases here:

- The router posts a buffer containing the nids for interested clients to read, with LNET_MD_MANAGE_REMOTE and LNET_MD_OP_GET flags set. All clients will get the buffer with offset as zero since they will get a complete list of nids on the router.

- For the case of an early server reply when adaptive timeout is in use, the client posts one reply buffer before sending out the request. The server first responds with an early reply with the offset set to zero. This means "I got your request, now wait patiently." Later, the server sends the actual reply to the same buffer with the proper offset.

Another type of attribute associated with the MD defines the operations allowed on the MD. For example, if the MD only has a get flag LNET_MD_OP_GET, then writing into this MD is not allowed. This is equivalent to the read-only case. And LNET_MD_

OP_PUT means the MD only allows for a PUT operation, but without the GET flags, it becomes a write-only.

There are two flags dealing with the case where an incoming message is bigger than the current MD buffer. The LNET_MD_TRUNCATE allows a save but truncates the message. This is useful when the client does not care about the content of the reply. As an example, let's assume a client pings the router to see if it is alive. The router replies with a list of its nids. The client does not care about the contents and does not interpret those nids, so a small buffer with a truncate flag is just fine. The second flag is LNET_MD_MAX_SIZE, which tells LNET to drop the message if the incoming message is beyond the allowed size.

Event Queue

Each MD has an event queue to which events are delivered. Each event queue can use callbacks or it can be polled.

9.2 Portal RPC: A Client of LNET

Portal RPC by design has multiple portals for each service defined, and these are *request*, *reply*, and *bulk* portals. We use an example to illustrate the point. Say a client wants to read ten blocks of data from the server. It first sends an RPC request to the server telling that it wants to read ten blocks and it is prepared for the bulk transfer (meaning the bulk buffer is ready). Then, the server initiates the bulk transfer. When the server has completed the transfer, it notifies the client by sending a reply. Looking at this data flow, it is clear that the client needs to prepare two buffers: one is associated with **bulk Portal** for bulk RPC, the other one is associated with **reply Portal**.

This is just how Lustre makes use of Portal RPC. It uses two portals in the above scenario because Portal RPC uses the same ME match bits for both. However, it is fine to have two buffers posted using the same portal as long as their ME matching bits are different.

Get and Put Confusion

- Most of the time, a client issues LNetPut to request something from the server or to request that something be sent to the server. Then the server will use LNetPut to send a reply back to the client or LNetGet to read something from the client as in bulk transfer.
- The one case that a client will issue LNetGet is the router pinger; the client will recieve a list of NIDs from a well-known server portal as a way of verifing that the router is alive.

Router In the Middle

We will use LNET PUT as an example to explain this scenario. Let's assume the client, *C*, has some payload to send to a server *S*, as the final destination through a router in the middle, *R*, as shown in Figure 13. LNET takes the payload and appends the PUT header with information such as final server NID, match bits, offset, etc. This is now a complete LNET message. Then the LNET passes this message on to the LND layer, along with the router NID. LND puts this message on the wire, and this message will be transmitted to *R*.

The router LND layer receives this message and passes onto the router LNET layer. Two pieces of information are needed from the PUT header in the message. One is the size of the message so that router can allocate (or pre-allocate) space to store the incoming message, noted that this space has nothing to do with the MD we discussed before, it is just a piece of buffer. The second piece of information is the destination NID. After the message is completely recieved, router LNET will pass this message along with destination NID to proper LND (it can be a different LND if this is heterogeneous routing environment) to send it on the wire.

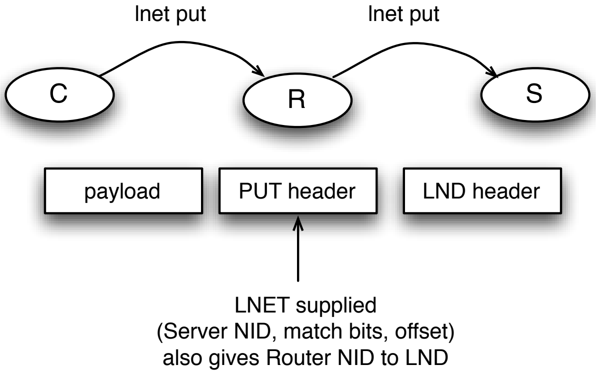
lnet put
lnet put
C
R
S
payload
PUT header
LND header
LNET supplied
(Server NID, match bits, offset)
also gives Router NID to LND

Figure 13: The illustration of LNET PUT with a router in the middle scenario.

9.2.1 Round 1: Client Server Interactions

Assume that a server wants to expose a buffer to its client; below are the steps happening on both sides, described briefly:

1. Server attaches the ME and MD and makes them part of some known portal.
2. Server makes sure that the MD option is set correctly: get-only, for remote management in this case. Now the server side is ready.
3. Client prepares the local ME and local MD for the reply if so requested.
4. Client invokes the LNET GET, with peer address, portal, match bits, etc.
5. Server receives the GET message, checks its validity (correct portal, correct match bits, correct operation on the MD).

6. Server invokes the callback registered on the MD to notify the upper layer. In the case of the router ping scenario we described earlier, once the buffer is posted, the server at the upper layer may not be interested in client requests, so this callback could just be NULL.

7. Server sends a reply message back to the client. Note that this reply is ***not*** a LNET PUT.

9.2.2 Round 2: More details

1. Server posts a request buffer by `ptlrpc_register_rqbd()`. The `_rqbd` is an abbrevation for request buffer descriptor, defined by the `struct ptlrpc_request_buffer_desc`. It holds enough information for creating the ME and MD. Since this is a server buffer that is to serve the request, the matching id, as well as all important ME, MD creation is as follows:

   ```
   struct ptlrpc_service *service = rqbd->rqbd->rqbd_service;
   static lnet_process_id_t match_id = {LNET_NID_ANY, LNET_PID_ANY};
   rc = LNetMEAttach(service->srv_req_portal, match_id, 0, ~0,
                       LNET_UNLINK,LNET_INS_AFTER, &me_h);
   rc = LNetMDAttach(me_h, md, LNET_UNLINK, &rqbd->rqbd_md_h);
   ```

   `LNET_UNLINK` indicates this ME should be unlinked when the MD associated with it is unlinked, and `LNET_INS_AFTER` means that the new ME should be appended to the existing list of MEs. The `me_h` and `md` are defined as `lnet_handle_me_t` and `lnet_md_t`, respectively.

2. Client sends an RPC by `ptl_send_rpc()`, and this function takes `struct ptlrpc_request *request` as an input parameter and *may* perform the following operations:

   - Client posts bulk buffer by `ptlrpc_register_bulk()`. This is performed when the bulk buffer is not NULL in the reqeust:

     ```
     if (request->rq_bulk != NULL) {
         rc = ptlrpc_register_bulk(request);
         ...
     ```

   - Client posts reply buffer by `LNetMEAttach()` and `LNetMDAttach()`. This operation is performed when a reply is expected (the input parameter `noreply` is set to 0).

   - Client sends request:

     ```
     rc = ptl_send_buf(&request->rq_req_md_h,
                       request->rq_reqmsg, request->rq_reqlen,
                       LNET_NOACK_REQ, &request->rq_req_cbid,
                       connection,
                       request->rq_request_portal,
                       request->rq_xid, 0);
     ```

3. Server handles incoming RPC by `request_in_callback()` defined in `events.c`. This can incur two further actions:

   - bulk transfer: `ptlrpc_start_bulk_transfer()`,

- send reply: `ptlrpc_reply()`.

4. Once the bulk transfer has been written, read, or replied to, there could be more callback invoked such as `client_bulk_callback()` and `reply_in_callback()`.

9.3 LNET API

Naming Conventions

Function names starting with `LNet` are external APIs for the upper layers. All other functions using lower cases are internal to LNET. Within LNET, LND has two sets of APIs. If it starts with `lnd`, then it is the API for LNET layer to call down or if it starts with `lnet`, then it is for LND to call up into LNET, as illustrated in the figure below. Note that it is confusing to use LNET to describe both as the whole network subsystem and as a particular layer paramount to LND, but that seems the way it is being used.

LNET compiles for both kernel space and user space. If a file is prefixed with `liblnet`, it is meant for user space; otherwise, it is for kernel space. Files under lnet/lnet are compiled twice, and those under `lnet/klnds` and `lnet/ulnds` are compiled only once.

```
   LNetPut      LNetGet
      |           |
+--------------------+
|        LNET        |
+--------------------+
|     |           |  |
|     v           v  |
| lnd_send   lnd_recv|
|                    |
|        LND         |
+--------------------+
```

Initialization and Teardown

`int LNetInit(void)` and `int LNetFini(void)` are two APIs for setting up and tearing down LNET connections.

Memory-Oriented Communication Semantics

The following API has been annotated with comments:

```
int LNetGet(
    lnet_nid_t        self,
    lnet_handle_md_t  md_in,        /* local MD to hold requested data */
    lnet_process_id_t target_in,    /* target process id */
    unsigned int      portal_in,    /* target portal index */
    __u64             match_bits_in,/* match bits used on target process */
    unsigned int      offset_in);   /* offset into remote MD */
```

This function initiates the remote read operation. Note that `offset_in` is only used when a target memory descriptor has the `LNET_MD_MANAGE_REMOTE` flag set. It is the same for the PUT operation.

```
int LNetPut(
    lnet_nid_t        self,
    lnet_handle_md_t  md_in,        /* local MD holding data to be sent */
    lnet_ack_req_t    ack_req_in,   /* flag to request Ack event */
    lnet_process_id_t target_in,    /* target process id */
    unsigned int      portal_in,    /* target portal index */
    __u64             match_bits_in,/* match bits for target process */
    unsigned int      offset_in,    /* offset into remote MD */
    __u64             hdr_data_in); /* 64-bit user data */
```

This function sends data asynchronously. The `self` specifies the source NID to use and hence the outgoing NI (network interface). If `LNET_NID_ANY` is given, LNet will choose source NID and NI by itself, based on destination NID and routing table. Note that acknowledgments are sent only when they are requested by the initiating process *and* the local MD has event queue *and* remote MD enables them.

Match Entry Management

```
int
LNetMEAttach(unsigned int portal,
    lnet_process_id_t match_id,
    __u64 match_bits, __u64 ignore_bits,
    lnet_unlink_t unlink, lnet_ins_pos_t pos,
    lnet_handle_me_t *handle)
```

This function creates a new ME. The first parameter indicates which local portal this ME should be associated with, and the next parameter indicates the remote process id (or remote peer) allowed to access this ME. The rest are match bits and ignore bits. Each portal RPC has a unique transaction id, so the portal RPC uses this transaction id as the match bits for the reply. The transaction id will be sent over to the remote peer, and the remote peer will use this transaction id as the match bits in its reply buffer. The last parameter is an ME pointer; if the call succeeds, it returns a handler.

```
int LNetMDAttach(
    lnet_handle_me_t meh, /* ME to be associated with */
    lnet_md_t umd,        /* user-visible part of the MD */
    lnet_unlink_t unlink, /* if MD is unlinked when it is not active */
    lnet_handle_md_t *handle)
```

This function is used to create an MD and attach it to an ME. An error returns if the ME already has an MD associated with it. `umd` comes from LNET client (Portal RPC or LNET self-test for now), and it specifies parameters for the to-be-created MD object, to which a handle will be returned in the `lnet_handl_md_t *handle`.

```
int LNetMDBind(
    lnet_md_t umd,
    lnet_unlink_t unlink,
    lnet_handle_md_t *handle)
```

This function creates a *standalone* memory descriptor, i.e., an MD that is not attached to an ME.

9.4 LNET/LND Semantics and API

LNET is connectionless, asynchronous, and unreliable. However, most LNDs are reliable and connection-oriented such that they need to establish a connection before they can talk to their peers. A LNET message has a payload restriction of 1MB, *and* the maximum number of segments cannot exceed 256. Furthermore, LNET does not fragment or assemble fragments. It's assumed that the upper layer never gives LNET a bigger payload than 1MB. There are several reasons for this limitation—for example the pre-set limit makes buffer management easier and some low-level driver has a limit on the number of scatter-gather buffers, such as 256. Also, an upper layer such as Portal RPC can fragment data more easily if buffers are posted in pages. The downside to this limitation is that if there is ever a network technology that supports an MTU more than 1MB, then LNET might not be able to make full use of its bandwidth. [19]

LND can have multiple instances—for example, in the cases where you have more than one IB interface. Each interface is represented by a network interface type defined by `lnet_ni_t`. One of the fields defined in that structure is `lnd_t`—a method table callable on this particular LND type.

9.4.1 API Summary

LND APIs are the interface between LNET and its underlying network drivers. As mentioned before, there are two sets of LND APIs. The first set is what LNET expects LNDs to implement. For example, LNET expects to call LND methods to send and receive messages.

- `lnd_startup()`, `lnd_shutdown()`: These functions are called per interface, whenever LNET wants to bring up or shut down the interfaces.
- `lnd_notify()`: This is optional.
- `lnd_accept()`: This is optional.
- `lnd_send()`, `lnd_recv()`, `lnd_eager_recv()`: Sends outgoing messages and receives incoming messages.
- `lnd_ctl()`: This call is optional. It passes user space `ioctl` commands to LND. LNET supports many ioctls through a special device file; some are directly handled by LNET (for example, adding a route) and others must be passed onto LND to handle.

The other set of APIs are LNET functions exported for LNDs to use:

- `lnet_register_lnd()` and `lnet_unregister_lnd()`: Each LND driver calls this function to register a new LND type.
- `lnet_notify()`: If a peer goes down, this function is called to notify LNET.
- `lnet_parse()`: For each new message received, LND calls this function to let LNET know that a message has just arrived, so that LNET can parse the message and check into it.

[19] MTU is used for allocating the network bandwidth fairly. For example, bigger MTU values might cause a malicious client to consume the network bandwidth unfairly, while other clients might be starving.

- `lnet_finalize()`: This is called by LND on both incoming and outgoing messages. When a message is sent, this call by LND allows LNET to generate an event to tell that the message has been sent. For an incoming message, this calls indicates the payload of the incoming message that has been completely received.

9.5 LNET Startup and Data Transfer

Brief notes on LNET code organizations:

```
lnet/ulnds      /* LND in user space, */
lnet/klnds      /* LND in kernel space */
lnet/lnet       /* LNET layer */
```

Apparently, there is not much code sharing between the kernel LND and the user space LND. Only Portals network and TCP network have an user space LND for that reason. In the kernel space, `ptllnd` [20], `o2iblnd`, and `socklnd` are probably the most important ones to know.

9.5.1 Startup

`lnet_startup_lndnis()` is invoked when LNET gets initialized.

1. Calls on `lnet_parse_networks()` to parse module parameters provided by the user. Afterwards, LNET gets a list of network interfaces to bring up.
2. Iterates each of the interfaces acquired above. First it tries to locate the lnd instance represented by `lnd_t`. Upon finding it, it invokes the `lnd_startup()`. We will look at each step in more detail.
3. Within the loop, it first looks for the driver by network type:

   ```
   lnd = lnet_find_lnd_by_type(lnd_type);
   ```

   If a driver is not found, then it could be that the driver has not been loaded yet, so it will try to load the module and then retry to locate the driver:

   ```
   rc = request_module(libcfs_lnd2modname(lnd_type));
   ```

   After the driver is loaded, during its module initialization, it registers with LNET, allowing the driver to be located later on.
4. After the driver for the network interface is located, then we can *bind* this driver to the interface and invoke the driver's startup method:

   ```
   ni->ni_lnd = lnd;
   rc = (lnd->lnd_startup)(ni);
   ```

Now we will explain a specific LND module, socket LND. In particular, we look into `ksocknal_startup()`.

[20] For Cray SeaStar system.

1. If the function is being called for the first time, then it invokes `ksocknal_base_startup()` to do some one-time initialization, for example, creating the data structure shared by all interfaces.

2. Finds out which Ethernet interface to use for this network. It can be user-specified or it can search for the first non-loopback interface in the system. Once located, it initializes the data structure for this interface. The nid is generated by:

```
ni->ni_nid = LNET_MKNID(LNET_NIDNET(ni->ni->nid),
    net->ksnn_interfaces[0].ksni_ipaddr);
```

After LNET is initialized, a user can send or receive data on this interface.

9.5.2 LNET Send

We describe the general flow of the send path, starting from `LNetPut` down to the LND layer.

1. First, the LNET message descriptor is allocated, `msg`, defined by `struct lnet_msg_t`. This message descriptor will be passed on to LND. In particular, there is a field within defined as `lnet_hdr_t msg_hdr` (the message header) which will eventually be a part of the on-the-wire message.

```
msg = lnet_msg_alloc();
```

2. LNET MD handle is converted from the incoming parameter to a real MD structure, which will hold the payload:

```
md = lnet_handle2md(&mdh);
```

3. The MD is associated with the message:

```
lnet_commit_md(md, msg);
```

4. Message details are filled in. These details can be the type of message (PUT or GET), match bits, portal index, offset, and user-supplied data, if any.

```
lnet_prep_send(msg, LNET_MSG_PUT, target, 0, md->md_length);
msg->msg_hdr.msg.put.match_bits = cpu_to_le64(match_bits);
msg->msg_hdr.msg.put.ptl_index = cpu_to_le32(portal);
...
```

5. The event information is filled in as given below.

```
msg->msg_ev.type = LNET_EVENT_SEND;
msg->msg_ev.initiator.nid = LNET_NID_ANY;
msg->msg.ev.initiator.pid = the_lnet.ln_pid;
...
```

6. Finally, lnet send function (not LND send) is invoked as below.

```
rc = lnet_send(self, msg);
```

7. This send function needs to locate the interface to use. If the destination is local, then it resolves to the directly attached interface. If the destination is remote, then it resolves the interface to a router from the routing table. The result of this search is `lp`, defined by `struct lnet_peer_t`, representing the best choice of a peer. This peer can be either the interface for the router or the interface for the final destination as below.

```
msg->msg_txpeer = lp;
```

8. Next, `lnet_post_send_locked()` is called for credit checking. Suppose you are only allowed to send *x* number of concurrent messages to the peer. If you exceed this credit threshold, the message will be queued until more credits are available.

9. If credit checking passed, then:

```
if (rc==0)
    lnet_ni_send(src_ni, msg);
```

This send function invokes LND send for further sending:

```
rc = (ni->ni_lnd->lnd_send)(ni, priv, msg);
```

At some later point, after LND finishes sending the message, `lnet_finalize()` will be called to notify the LNET layer that the message is sent. However, let's continue down the path of sending data. Let's assume it is an IP network; then socket LND send, more specifically, `ksocknal_send()` will be called.

10. Remember that socket LND is connection based, so when you want to send something, first you need to locate the peer, then you have to check if a connection has been established. If there is one, you just queue the `tx` to the connection.

```
if (peer!=NULL) {
    if (ksocknal_find_connectable_route_locked(peer) == NULL) {
        conn = ksocknal_find_conn_locked(tx->tx_lnetmsg->msg_len, peer);
        if (conn != NULL) {
            ksocknal_queue_tx_locked(tx,conn);
            ...
 }
```

So eventually, the message queued will be sent on the socket connection through the kernel socket API.

11. If there is no connection yet, then we queue the message to peer first, so that when a new connection is established, we can move the message from the peer's queue to the connection's queue to be sent out.

The message format sent on the wire has the following layout, briefly:

```
 +--------------+-----------+-----------------+-------------+-----------+
 | TCP, IP, and | sock LND  | LNET msg type   | LNET common |  payload  |
 | MAC header   | header    | specific header | header      | (from MD) |
 +--------------+-----------+-----------------+-------------+-----------+
```

12. There are two ways (APIs) to send a message from socket LND: If the message is small, normal send will put the message into a kernel socket buffer (each socket has a send buffer). This is *not* a zero-copy transfer. Alternatively, you could send this buffer directly onto the network without making a copy first (zero copy transfer). However, zero-copy has its own overhead, so Lustre only utilizes this path for large messages.

9.5.3 LNET Receive

On the receiving end, assuming that we are using socket LND, `ksocknal_process_receive()` is the entry function for receiving. The general steps are given here.

1. LND starts to receive the new message. At first, it only receives up to LNET message header because it doesn't know where to put the payload yet, only LNET layer has the information on the destination MD.

   For this reason, LND gives LNET header to `lnet_parse()`. This function LNET layer will look into the header and identify the portal, offsets, match bits, source NID, source PID, etc. LNET can either reject the message (for example, malformed) or accept it.

2. If proper MD is located, LNET (from `lnet_parse()` calls another LND API, `lnd_recv()`. Now the payload from the socket kernel buffer is copied over to the destination MD. In the case of socket LND, this is one kernel memory-to-memory copy.

3. LNET calls `lnd_eager_recv()` immediately if it will call `lnd_recv()` sometime later.

4. After LNET calls `lnd_recv()`, LND starts receiving the payload (either by memory-to-memory copy or RDMA) and LND should call `lnet_finalize()` within a finite time. LND could use RDMA for data transfer at this point if it is available.

Also note that TCP can perform fragmentation, but when `lnd_recv()` finishes, it delivers the complete message, after de-fragmenting the message.

9.5.4 The Case for RDMA

We have mentioned that there is a memory-to-memory copy in socket LND. For any RDMA supporting network, for example, o2ib LND, it would RDMA data directly into destination MD, therefore avoiding the memory-to-memory copy. More complete interactions are as follows:

1. LNET PUT passes the message to o2ib LND. Now o2ib LND has two pieces of information: An LNET message header, which holds information such as source NID and destination NID, and an MD pointing to the actual payload.

2. Unlike socket LND, o2ib LND only sends (using OFED API) the LNET header over the network to the peer. The message includes an o2ib LND header, which indicates this is an o2ib PUT request.

3. At the receiving end, o2ib LND still calls `lnet_parse()` as it has the LNET header information and it identifies the MD. Then it calls `lnd_recv()` to receive the data. o2ib's `lnd_recv()` will register the MD buffer with OFED first. After the memory is registered, it gets an OFED *memory ID*, which identifies the registered memory.

4. Now o2ib sends another message (PUT ACK) back to the initiator, along with the remote memory ID. The initiator registers this message with its local MD holding the payload and gets a local memory ID. Finally, it calls on `RDMA write` and passes the control to OFED for further processing.

9.6 LNET Routing

General Concepts

There are two basic characteristics about Lustre routing. First, all nodes inside a Lustre network can talk to each other directly without any involvement from the routing layer. Second, Lustre routing is static routing, where its network topology is statically configured and parsed when the system initializes. Configuration can be updated at the runtime and the system responds to it, however, this "dynamic update" is very different from distance vector or link state based routing as we know it.

A crude definition of a Lustre network would be *a group of nodes that can communicate with each other directly without involvement of routing*. Each Lustre network has a unique type and number, for example, tcp0, ib0, ib1, etc., and each end node has an NID (Network Identifier). Figure 9.6 illustrates a sample LNET routing layer stack. Here are some artifacts of this definition: (1) It has nothing to do with IP subnet and IP router. So if you have two networks that have an IP router in between, it *can* still be considered one Lustre network. (2) If you have one end node with TCP interface and another end node with IB interface, then you must have an LNET router in between, and this will be considered two Lustre networks. (3) The addresses within a Lustre network must be unique.

Another implication of the above definition is that the routing table of an end node will have LNET router as its next hop, not the IP router. To specify a Lustre network, you typically use one of two directives: `networks` or `ip2nets`:

```
# router
options lnet networks = tcp0(eth0), tcp1(eth1)
# client
options lnet networks = tcp0
# single universal modprobe.conf
options lnet ip2nets="tcp0(eth0,eth1) 192.168.0.[2,4]; tcp0 192.168.0.*; \
    elan0 132.6.[1-3].[2-8/2]"
```

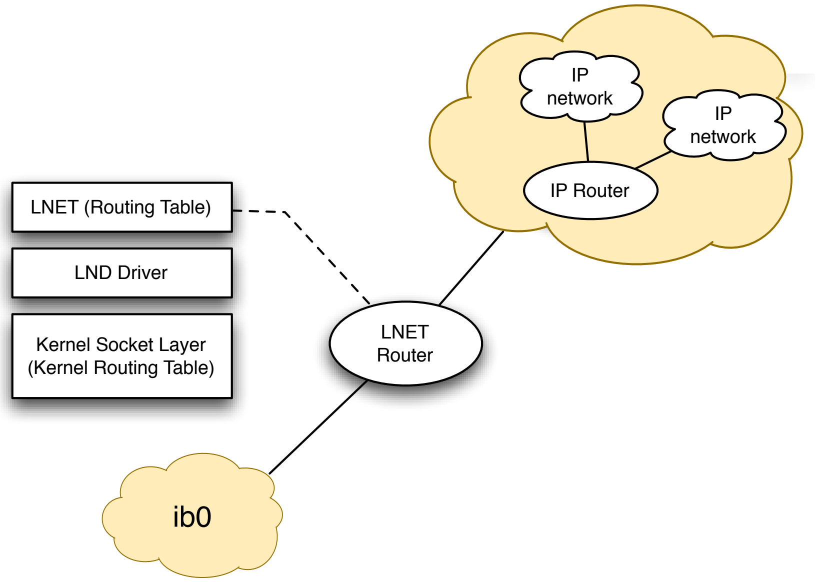

Figure 14: Illustration of LNET routing layer stack.

9.6.1 Asymmetric Routing Failure

This topic highlights a deficit we discovered in the current LNET production code. It is pertinent to the general discussion on LNET, and we believe it would be good to share our observation with a broader audience. The problem is that it can be difficult for a router to reliably determine if an interface is down. Right now, LNET does not even try to do it. So for a configuration where a router has two interfaces, such as `tcp0` and `ib0`, if the `ib0` interface is down and end nodes connected with `tcp0` are still pushing data, this will result in an intermittent communication failure.

One idea towards solving this problem is that the LNET router can try to detect the transmission problem, then temporally disable all its interfaces. A router might be able to do this more intelligently by disabling only some interfaces if it has information on the full topology and the incoming path that clients use. Until then, shutting down all of its interfaces seems to be better than suffering timeout as a result of intermittent communication failure.

9.6.2 Routing Buffer Management and Flow Control

Upon initialization, an LNET router pre-allocates a fixed number of routing buffers. The LNET layer of an end node won't need to do this (except when initializing its routing tables). Allocating and managing routing buffers are the primary, if not the only, difference between the execution logic of an end node and a router.

As routing buffers are limited resources, to prevent a single end node from overwhelming a router, each end node is given a quota. Once a buffer has been forwarded, it can be recycled again. The quota for each end node is also known as **buffer credit**. It is only significant for routers, and it is different from the “RPCs in flight” credit, which is known as **peer to peer credit**. One RPC can involve several LNET messages (at most ten), and these can be one request LNET message, one reply LNET message, and in the case of a bulk transfer, there could be four LNET messages going one way, and four LNET messages going in the opposite direction. This also implies that the bulk transfer LNET messages are part of the bulk RPC transaction, but they are **not** considered an RPC. Therefore, they (the four bulk transfer messages) are not counted towards RPCs in flight.

There are three kinds of router buffers: 1MB (maximum amount of payload an LNET message can carry), 4KB for small messages, and zero payload buffers for tiny messages such as ACK.

Requests from an end node are on a first come, first served basis. If requests from a particular end node exceed its quota, then its next request will not be handled until the router buffer is freed, which is essentially the way LNET layer does flow control (flow control is point-to-point). It is up to the upper layer to implement an end-to-end flow control, for example, by number of RPCs in flight. The buffer can be freed when the underlying LND calls the `lnet_finalize()` as an indication that LND considers the forwarding done, but it doesn’t really mean the message has been put on the wire. Depending on the size of the message, LND and kernel may have different logics for handling it. The only requirement from LNET is, once `lnet_finalize()` is invoked, LNET should be able to recycle the buffer freely.

Logically, LNET has a single queue, and incoming messages from all interfaces are queued and processed in order. Each interface has its own interface queues; however, that is not a concern of LNET since this is interrupt-driven. So it is guaranteed that each incoming message is handled in the order it arrives.

9.6.3 Fine Grain Routing

This feature is a recent development (Lustre bug #15332) for Jaguar/Spider deployment at ORNL. The motivation is that LNET doesn’t assign weights to routes. So if you have multiple routers that reach the same destination, LNET will perform a round robin algorithm to distribute the load, and this applies to both end node and router. What fine grained routing adds, is simply, to assign weights to different routers, preconfigured by the system administrator. The goal is that *better* routes gets more traffic. Here, better is defined by the site system administrator.

```
dest network 1:
     w1    (router 1, router 3)
     w2    (router 4, router 5)
     w3    (router 2)
```

So for example, you can specify different weight classes and assign routers to each to indicate your preference. If $w_1 < w_2$, then w_1 is the preferred weight class of the two. Within a given weight class, routers are equal.

More specific to our case, this mechanism provides the potential for a client to pick a router closer to its proximity as its preferred router.

10 Lustre Generic Filesystem Wraper Layer: fsfilt

Lustre provides a generic wrapper layer named *fsfilt* to interface between the underlying local filesystem and Lustre. The upper layer, `obd_filter`, uses generic functions provided by the `fsfilt` layer, and then `fsfilt` layer passes these calls into a filesystem specific implementation. These specific implementations are interfaces to the particular underlying filesystem. The fsfilt calls the local filesystem by using tiny wrappers targeted for the particular filesystem (e.g., `fsfilt_ext3` for *ext3* filesystem and `fsfilt_reiserfs` for *Reiserfs3* filesystem). This section outlines details of the `fsfilt` layer and analyzes `fsfilt_ext3` as an example interface implementation.

10.1 Overview

The `fsfilt` framework is largely defined by the `lustre/include/luste_fsfilt.h` file. In this file `struct fsfilt_operations` defines operations required from the underlying filesystem. At the `obd_filter` registration time, the system tells what filesystem it is built under and Lustre calls the required filesystem registration operations. This is done during the kernel module (e.g., `ldiskfs`, `fsfilt_ext3`) initialization phase. Corresponding source code for each filesystem implementation is located in `lustre/lvfs`. However, the `fsfilt_ldfiskfs.c` file will be missing with the *HEAD CVS* checkout, because it is generated at build time from `ldiskfs_ext3.c` (same as in other CVS branches) by taking `fsfilt_ext3.c` and replacing all *ext3* occurrences with *ldiskfs*. Figure 15 shows an example implementation of the Lustre `fsfilt` layer components for Linux and their communication paths.

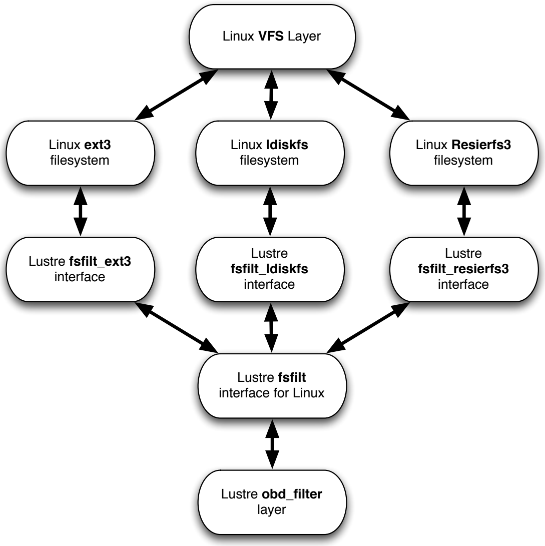
Linux VFS Layer
Linux ext3 filesystem
Linux ldiskfs filesystem
Linux Resierfs3 filesystem
Lustre fsfilt_ext3 interface
Lustre fsfilt_ldiskfs interface
Lustre fsfilt_resierfs3 interface
Lustre fsfilt interface for Linux
Lustre obd_filter layer

Figure 15: An example implementation of the Lustre fsfilt layer.

One interesting point to mention is that although in the lustre/lvfs/fsfilt.c file there are defined symbols as below:

```
EXPORT_SYMBOL(fsfilt_register_ops);
EXPORT_SYMBOL(fsfilt_unregister_ops);
EXPORT_SYMBOL(fsfilt_get_ops);
EXPORT_SYMBOL(fsfilt_put_ops);
```

this file is not used for creating the fsfilt kernel module; in fact, there is no such kernel module. It is linked with lvfs module to provide an API to register fsfilt methods to access specific filesystems. The get_ops and put_ops calls allow those who use fsfilt services to get pointers to appropriate operation tables by names and also to get a pointer and a reference, so that it is not allowed to unload the fsfilt_* module that is in use. When these functions are stopped being used, they release respective pointers.

The list below shows struct fsfilt_operations methods defined in the lustre/include/linux/lustre_fsfilt.h file.

```
struct fsfilt_operations {
        struct list_head fs_list;
        struct module *fs_owner;
        char   *fs_type;
        char   *(* fs_getlabel)(struct super_block *sb);
        int     (* fs_setlabel)(struct super_block *sb, char *label);
        char   *(* fs_uuid)(struct super_block *sb);
        void   *(* fs_start)(struct inode *inode, int op, void *desc_private,
                             int logs);
        void   *(* fs_brw_start)(int objcount, struct fsfilt_objinfo *fso,
                                 int niocount, struct niobuf_local *nb,
                                 void *desc_private, int logs);
        int     (* fs_extend)(struct inode *inode, unsigned nblocks, void *h);
        int     (* fs_commit)(struct inode *inode, void *handle,int force_sync);
        int     (* fs_commit_async)(struct inode *inode, void *handle,
                                        void **wait_handle);
        int     (* fs_commit_wait)(struct inode *inode, void *handle);
        int     (* fs_setattr)(struct dentry *dentry, void *handle,
                               struct iattr *iattr, int do_trunc);
        int     (* fs_iocontrol)(struct inode *inode, struct file *file,
                                 unsigned int cmd, unsigned long arg);
        int     (* fs_set_md)(struct inode *inode, void *handle, void *md,
                              int size, const char *name);
        int     (* fs_get_md)(struct inode *inode, void *md, int size,
                              const char *name);
        int     (* fs_send_bio)(int rw, struct inode *inode,struct kiobuf *bio);
        ssize_t (* fs_readpage)(struct file *file, char *buf, size_t count,
                                loff_t *offset);
        int     (* fs_add_journal_cb)(struct obd_device *obd, __u64 last_rcvd,
                                      void *handle, fsfilt_cb_t cb_func,
                                      void *cb_data);
        int     (* fs_statfs)(struct super_block *sb, struct obd_statfs *osfs);
        int     (* fs_sync)(struct super_block *sb);
        int     (* fs_map_inode_pages)(struct inode *inode, struct page **page,
                                       int pages, unsigned long *blocks,
                                       int *created, int create,
                                       struct semaphore *sem);
        int     (* fs_write_record)(struct file *, void *, int size, loff_t *,
                                    int force_sync);
        int     (* fs_read_record)(struct file *, void *, int size, loff_t *);
        int     (* fs_setup)(struct super_block *sb);
        int     (* fs_get_op_len)(int, struct fsfilt_objinfo *, int);
        int     (* fs_quotacheck)(struct super_block *sb,
                                  struct obd_quotactl *oqctl);
        __u64   (* fs_get_version) (struct inode *inode);
        __u64   (* fs_set_version) (struct inode *inode, __u64 new_version);
        int     (* fs_quotactl)(struct super_block *sb,
                               struct obd_quotactl *oqctl);
        int     (* fs_quotainfo)(struct lustre_quota_info *lqi, int type,
                                 int cmd);
        int     (* fs_qids)(struct file *file, struct inode *inode, int type,
                           struct list_head *list);
        int     (* fs_get_mblk)(struct super_block *sb, int *count,
                               struct inode *inode, int frags);
        int     (* fs_dquot)(struct lustre_dquot *dquot, int cmd);
        lvfs_sbdev_type (* fs_journal_sbdev)(struct super_block *sb);
};
```

10.2 fsfilt for ext3

As mentioned above, Lustre provides a special `fsfilt` implementation for every underlying local filesystem. This section explains `fsfilt_ext3` implementation for the Linux ext3 filesystem. The `lustre/lvfs/fsfilt_ext3.c` file is used for declaring the `fsfilt` implementation for ext3. Upon build, `fsfilt_ext3` is plugged into the kernel as a module. The entry point for the module is `module_init(fsfilt_ext3_init)` and the exit point is `module_exit(fsfilt_ext3_exit)`.

```
static int __init *fsfilt_ext3_init(void) {}
```

is used for initialization at the registration time. All local filesystem specific `fsfilt` implementations have this call.

Within the `init` function, first a cache is created by `cfs_mem_cache_create()` for the callbacks for journal commits. This allows a callback to happen when a certain journal transaction is committed. Currently, Lustre does not support any underlying filesystems without a journal. The return value for the cache creation in the `init` method is `fcb_cache` variable. Here, *fcb* denotes commit callback data.

Also in the `init` method, the same as any other fsfilt implementation, `fsfilt_ext3.c` declares the permitted operations for that particular underlying filesystem by providing a one-to-one mapping between the `fsfilt` methods and `fsfilt_ext3` operations through the `static struct fsfilt_operations fsfilt_ext3_ops={}` definition. Some important `fsfilt_ext3` methods are explained in more detail below.

```
static void *fsfilt_ext3_start()
```

starts the journal for metadata operations. It checks what kind of operation is called for through the switch at the beginning of this function call. For each operation it calculates the maximum number of blocks required for that particular metadata transaction (denoted by `nblocks`). For ext3, when a metadata transaction is initiated, it is required to identify how many blocks will be used for that transaction. If the requested number of blocks is less than the available number of blocks, the filesystem will flush some number of blocks to accommodate the requested block size. This functionality is not required for a filesystem like ZFS.

```
static char *fsfilt_ext3_get_label()
```

gets the filesystem label, while

```
static char *fsfilt_ext3_set_label()
```

sets the filesystem label.

```
static char *fsfilt_ext3_uuid()
```

allows `fsfilt` layer to query the filesystem Linux UUID.

```
static void *fsfilt_ext3_brw_start()
```

starts a journal transaction for the block I/O operation. It takes a buffer of pages as an argument.

Flow wise, the metadata operations call `*fsfilt_ext3_start()` to open a journal transaction and do metadata operation, while the block I/O operations call `*fsfilt_ext3_brw_start()` to open a journal transaction. However, `*fsfilt_ext3_brw_start()` does not really perform the block I/O but, rather, generates a journal handle for the transaction. For both operations the journal is identified by the `*journal` pointer, which is of type `journal_t`.

Also, as can be seen in the source code, both function calls have a `*handle` pointer which is of type `handle_t`. This is used for any operation that requires a journal handle. This handle will be passed as an argument to its function call.

An important point to mention is that in Lustre when I/O is done to a file, the inode is modified first and all the required blocks are allocated, then the transaction is closed and finally the I/O is performed separately, so that the journal transaction is not kept open for the whole duration of the actual I/O.

```
static int fsfilt_ext3_commit()
```

commits and closes the current open journal transaction. This function also has a flag called `force_sync`, which signals whether the flush should be ***immediate***. `force_sync=0` means just close the transaction and ***do not immediately*** flush, whereas `force_sync=1` means close the transaction and ***immediately*** flush the memory copy.

```
static int fsfilt_ext3_commit_async()
```

also closes the transaction and returns a handle (`**wait_handle`) to the caller to be used later when actually committing the transaction.

In the true sense of these two operations, `fsfilt_ext3_commit` and `fsfilt_ext3_commit_async` are both asynchronous. However with the availability of the `force_sync` option, the `fsfilt_ext3_commit` operation has the possibility of synchronously committing a given journal transaction. Most commonly, `fsfilt_ext3_commit` function is used whereas `fsfilt_ext3_commit_async` is used exclusively for the `DATA` mode.

```
static int fsfilt_ext3_send_bio()
```

submits the I/O to the file. The I/O is already formed before calling this function such that a list of buffer is created and their destination on the disk is set in the kernel. This function is used only for file data, so there is no need to open a journal transaction at this time. By the time this function is called, the journal transaction is already closed for performance reasons. The typical flow for a block I/O is discussed in Section 10.3. This flow is followed by a `fsfilt_ext3_send_bio` function call for performing the actual block I/O. For Linux 2.6 kernels all this function does is to call the `submit_bio()` kernel function. The procedure is more complicated for Linux 2.4 kernels, and it is beyond the scope of this report.

```
static int fsfilt_ext3_extend()
```

extends the journal transaction by the denoted number of blocks. The number of extra blocks requested is denoted by the `unsigned nblocks`. After the extension the transaction will still have the same handle denoted by the `*handle` as Lustre maintains a single transaction visibility. This is especially useful when it is impossible to predict how many blocks an operation will take. Examples where this functionality will be useful are truncate or unlink operations.

```
static int fsfilt_ext3_setattr()
```

updates the inode size and attributes (user group, access mode, various file times and file sizes).

```
static int fsfilt_ext3_iocontrol()
```

passes the iocontrol (or ioctl) parameters to the filesystem below.

```
static int fsfilt_ext3_set_md()
```

and

```
static int fsfilt_ext3_get_md()
```

are used for setting and querying the striping information. For ext3 this information is kept in EA, and the implementation is filesystem specific.

```
static size_t fsfilt_ext3_readpage()
```

is for reading a page from the underlying file or directory. The switch at the begining of this function determines if it is a file or a directory to be read. If it is a normal file, then it simply calls a `read()` kernel function to fill in the buffer with the file data information. However, if it is a directory to be read, then it calls ext3 directory read functions to read from the directory.

```
static int fsfilt_ext3_add_journal_cb()
```

is used for updating the in-memory representation of what is actually committed on disk on a server from a transaction point of view. This is useful when replying to a client with the in-memory representation of what is actually committed on disk on that particular server for that particular client, so that the client can discard all data up to that given transaction number to save memory. The advantage of having this functionality is that, in case of a server crash, the data will still be in the client's memory since the server wouldn't have responded back with the `*cb_data` yet. Here, `*cb_data` holds the pointer address for the number of the last committed transaction on the disk. The drawback is that by keeping an extra copy on the client (besides the copy on the server), memory consumption is increased.

```
int fsfilt_ext3_map_ext_inode_pages()
```

is the function that allocates the blocks. It is used to get information about blocks underlying a certain block range and if they are not allocated, then it allocates them if requested. It sends the requestor an array of pages where the blocks are placed.

```
static int fsfilt_ext3_write_record()
```

and

```
static int fsfilt_ext3_read_record()
```

are for journal update of the special `LLOG` (Lustre Log) file. This file is automatically updated. The `LLOG` is required while performing an operation on multiple nodes (e.g., unlink, which has to start on the MDS and then continue on the OSTs).

```
static int fsfilt_ext3_setup()
```

is used when the ext3 filesystem is first mounted. The `obd_filter` initialization calls this function.

```
static int fsfilt_ext3_get_op_len()
```

is to get the number of blocks in the journal that a particular operation will require. This function is obsolete and not called at all.

```
static __u64 fsfilt_ext3_get_version()
```

and

```
static __u64 fsfilt_ext3_set_version()
```

are for setting and querying the inode version. Inode version is currently not used in the code base we have discussed so far. However, in future code bases, it will be used for version based recovery mechanisms.

10.3 fsfilt Use Case Examples

Typical flows for metadata operations, asynchronous block I/O operations, and synchronous block I/O operations can be seen in Figures 16, 17, and 18, respectively.

Figure 16 is an example of a metadata operation. This type of flow is used in Lustre either when there is no file data or when no client is available for replying to a certain set of metadata operations.

10.3.1 `DIRECT_IO` in Lustre

Lustre uses `DIRECT_IO` for all file data I/O operations from the `obd_filter` to disk and back (not cached at any point). The metadata is always journaled.

As can be seen from Figures 17 and 18, these flows guarantee that the data is actually written to the disk before the journal transaction is committed. These flows represent the `ORDERED` journal mode with `DIRECT_IO`. In the `DATA` mode, the bulk data is written to the journal first and then after it is committed to the journal, file data is transferred to the disk.

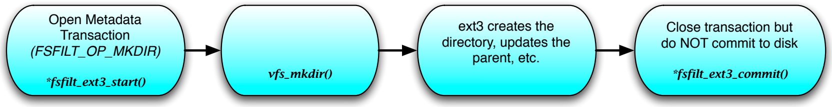
Open Metadata Transaction
(FSFILT_OP_MKDIR)
*fsfilt_ext3_start()
vfs_mkdir()
ext3 creates the directory, updates the parent, etc.
Close transaction but do NOT commit to disk
*fsfilt_ext3_commit()

Figure 16: Fsfilt flow path for metadata operations.

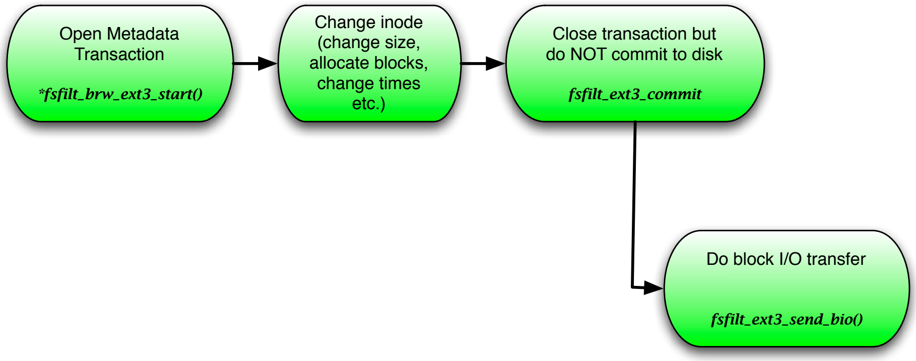
Open Metadata Transaction
*fsfilt_brw_ext3_start()
Change inode (change size, allocate blocks, change times etc.)
Close transaction but do NOT commit to disk
fsfilt_ext3_commit
Do block I/O transfer
fsfilt_ext3_send_bio()

Figure 17: Fsfilt flow path for asynchronous block I/O operations.

10.3.2 Replaying Last Transactions After a Server Crash

As an example, let's consider the following. Client connects and performs some operations on a Lustre server. Server replies back to the client with a commit, but at this point the data is actually not committed to the disk. Client now has the record of the last transaction being committed as $x+y$ while according to the server disk, it is just x. If the server crashes at this point, there is a conflict between the server disk and client in terms of the last committed transaction number. Let's further assume that following this chain of events, the server is rebooted and the client reconnects to the server. The server then replies back with acknowledgment and the number of the last transaction from that client that is already committed to the server disk. For our example this number is x. The client then replies back with $x+5$ being the last transaction number committed by that server and a list of the five last missing transactions to be performed by the server.

10.3.3 Client Connect/Disconnect

Another use case example for `fsfilt_ext3_commit` with `force_sync = 1` in Lustre would be the client connect/disconnect case. Each Lustre server maintains a separate table of client connect/disconnect operations. This table is kept on a separate file on each server and is journaled. When a client connects, it is written to a separate file and this file is fully journaled. To maintain this file, `fsfilt_ext3_commit` is used and the `force_sync` flag is set to 1, as there would be no client to reply after a

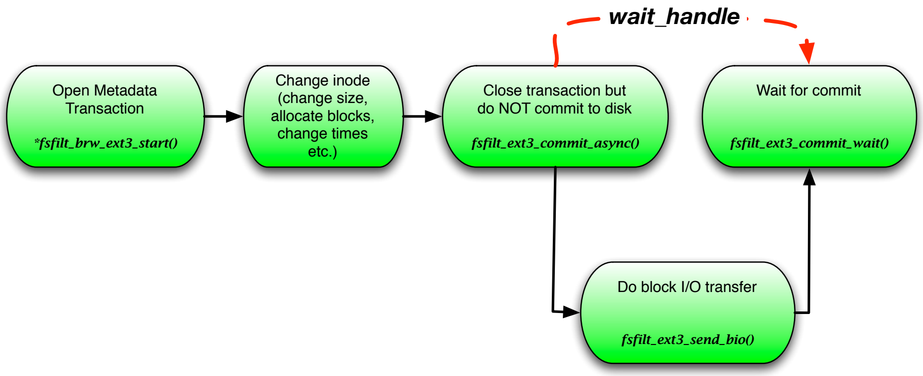
wait_handle
Open Metadata Transaction
*fsfilt_brw_ext3_start()
Change inode (change size, allocate blocks, change times etc.)
Close transaction but do NOT commit to disk
fsfilt_ext3_commit_async()
Wait for commit
fsfilt_ext3_commit_wait()
Do block I/O transfer
fsfilt_ext3_send_bio()

Figure 18: Fsfilt flow path for synchronous block I/O operations.

replay if the client is already disconnected.

10.3.4 Why `ls` Is Expensive on Lustre

As an example let's consider the `ls` operation on an ext3 based Lustre filesystem which is actually performed using the `fsfilt_ext3_readpage` behind the scenes. From the client perspective, it connects to the MDS and gets the content and inode number for that particular directory it is interested in. Based on the content information, the client then sends `stats` requests for each file in that directory back to the MDS. The MDS will reply back with striping information for each stat request (as well as with other pertinent file metadata per request). All of these are completed in separate RPCs. If, for example, there are three files in that given directory, by the end of this step, the client would have sent five separate RPCs to the MDS (dir open, readdir, and stats for every file). (In fact, regular `ls` does not need the file mode during `ls`, but most Linux distributions are now delivered with "color ls" that wants to output file information in color depending on file type, so it does a `stat()` call for every file.) Following this step, if the client is interested in file size data, as an example, it has to query all the OSTs that were listed in the stat information by sending glimpse RPCs. This step has to be repeated for every file separately. Again, as an example, if each of those three files are striped over 100 OSTs, by the end of this step, client would have sent $3+(3\times 100)$, or in other words, 305 RPCs. This is true not only for the `ls -l` but also for the regular `ls` case, such that it needs to find the file mode.

11 Lustre Disk Filesystem: ldiskfs

The ldiskfs (also sometimes wrongly dubbed the Linux ext4 filesystem) is a heavily patched version of the Linux ext3 filesystem and is developed and maintained by Sun Microsystems. ldiskfs is a superset of Linux ext3 and ext4 filesystems. Currently, it is only used by Lustre on servers as an underlying local filesystem. This section provides a brief overview of ldiskfs and its differences from the ext3 filesystem.

The main differences between the ext3 and ldiskfs filesystems in terms of their respective I/O paths are listed here.

- In ldiskfs, allocation/block lookup is done up front with inode lock held, but then inode lock is dropped while the I/O is being submitted to disk. This greatly improves I/O concurrency because inode is locked for only a short time.
- In ldiskfs, allocation is done for the entire RPC (1MB, or in other words, 256×4KB blocks) at one time, instead of 4KB block-at-a-time like VFS does. This avoids lots of overhead in repeated bitmap searching and also allows more efficient extent finding.
- In ldiskfs, writes are currently fully synchronous to the client (no reply until data and metadata are on disk), but this is changing. The newly developed "async commit" patch will reply to client after data is on disk, but before metadata is committed in any transaction. Also, the new ROC (1.8) will put write pages into cache before flushing them to disk, in preparation for a fully asynchronous write cache on the OSS.
- In ldiskfs, currently journal flush is forced after every write RPC.

In ldiskfs, as with any other underlying local filesystem to be used by the Lustre servers, the policies listed here can affect the I/O path.

- *Client-controlled RPCs in flight*, *RPC size* and *cached dirty data per OST*. *RPC size* is preferred to be a multiple of the underlying RAID stripe size to avoid read-modify-write operations. Also, $RPC\,size \times num\,of\,RPCs\,in\,flight \times num\,of\,clients$ should take into account the bandwidth of the link and bandwidth of underlying block devices to avoid big latency and requests piling up at contention points.
- *Server-controlled number of I/O threads*. This currently limits the amount of I/O to backend devices in total at any given time.
- *Striping policy*. It determines how many OSTs participate in I/O to a given file.

11.1 Kernel Patches

The ldiskfs and Lustre require a set of Linux kernel patches that not only adds new features on top of ext3 but also improves the performance. The following list provides the essential set of kernel patches for ldiskfs, although some might not be needed for certain use cases (e.g. `sd_iostats`).

- `jbd-2.6.10-jcberr.patch`: journal callback patch that allows replying to clients with the `last_committed` when data is committed at least to the journal. This also means that the data is recoverable by the local filesystem without the client support.

- `jbd-stats-2.6-sles10.patch`: provides statistics about transaction size, time, etc.

- `iopen-misc-2.6.12.patch`: allows `open-by-inode` for `getattr-by-fid` and recovery.

- `dev_read_only-2.6-fc5.patch`: discards writes to block devices for testing and *failback* of OST/MDT without blocking on in-flight I/O.

- `sd_iostats-2.6-rhel5.patch`: provides optional SCSI device statistics.

- `blkdev_tunables-2.6-sles10.patch`: allows submitting larger I/Os through the block layer.

- `i_filter_data.patch`: allows hooking MDS/OSS data from the in-memory inode.

- `quota-fix-oops-in-invalidate_dquots.patch`: fixes bug in upstream kernel quota code. This is fixed in the 2.6.17 kernel.

- `jbd-journal-chksum-2.6-sles10.patch`: provides journal transaction checksums to detect on-disk corruption.

11.2 Patches: ext3 to ldiskfs

Besides the patches listed above, ldiskfs requires a number of patches on top of the ext3 source base for increased performance and functionality. The following list shows the patches required for a ldiskfs filesystem.

- `ext3-wantedi-2.6-rhel4.patch`: allows creating specific inodes by number and is used for recovery. Suppose the client did an open/create and got a reply from the server with a specific inode number. Let's assume the server crashes at this point but the creation is not reflected to disk yet, so the information is lost. After the server comes back up and enters recovery, the client connects back and sends its open/create request back again, specifying the earlier communicated inode number from the server. In Lustre 2.0 this will not be needed, as there will be a different set of Lustre inode numbers than the actual respective inode numbers on disk. On servers a functionality for mapping Lustre inode numbers to actual on disk inode numbers will eliminate the dependency on this patch.

- `iopen-2.6-fc5.patch`: allows lookup inodes by number. Typically, one needs the full path to look up a file, but Lustre does not have this functionality in its protocol. When one does a lookup, it only has the parent inode number and child name, and normally this is enough. But if Lustre is in recovery mode, all previous lookup steps are lost and Lustre needs a method to find the parent inode number. This patch provides this functionality. This problem does not occur in ext3 because the filesystem is local, and at the time of a crash, everything goes down, and at the time of recovery (or reboot), all the look up functionality starts from the root directory.

- `ext3-map_inode_page-2.6-suse.patch`: allocates data blocks for bulk read/write operations for later submission for I/O. This patch creates an API that can be used to map `inode_page` function in the Lustre `fsfilt` layer. Lustre needs this patch because it allocates blocks ahead of time to enhance the metadata performance.

The next five patches can be grouped together as they provide `extent` format capability on top of the ext3 filesystem. They are also used in the Linux ext4 filesystem. The ext3 filesystem address files in blocks, and every block has a 4 byte record in inode with the block number where the actual data for this block is stored. For long files, this results in storing lots of block information in the inode, which is not very efficient. The `extent` concept defines a range of blocks to be used as a single entity in block allocation for large files. The maximum for a range of blocks in an `extent` is 128MB. This is due to the fact that for every 128MB there is a block holding the bitmap of the next 128MB of allocation. The only exception for this is files with holes, where a hole, no matter how big it is, can be represented with a single `extent`. For a very large file in Lustre with the following set of patches, one will have a list of `extents`, with each `extent` being 128MB less 4KB for the bitmap mapping for that particular 128MB `extent` group. The `extent` concepts improves the performance such that, not only at the time of allocating but, for example, when unlinking a file, one only needs to provide the block information defining the `extent` for that particular file.

- `ext3-extents-2.6.16-sles10.patch`:
- `ext3-extents-fixes-2.6.9-rhel4.patch`:
- `ext3-extents-multiblock-directio-2.6.9-rhel4.patch`:
- `ext3-extents-search-2.6.9-rhel4.patch`:
- `ext3-extents-sanity-checks.patch`:

Next two patches provide a multi-block (`mballoc`) allocator to the ldiskfs filesystem. They are also used in the Linux ext4 filesystem. Just to clarify at this point, `extent` and `mballoc` allocations are orthogonal to each other. When allocating an `extent` a contiguous set of blocks is allocated. Let's assume the first block in this `extent` allocation starts from x and the last block is $x+y$. The next `extent` to be allocated for the same file might or might not start from $x+y+1$, but in any case, this new `extent` will be allocated to a new set of contiguous blocks. This way, while addressing these two perhaps disjoint sets of blocks on disk, we only need to provide their `extent` information instead of addresing them block by block. `mballoc`, on the other hand, is a way to allocate the requested amount of (hopefully contiguous) blocks with one call.

- `ext3-mballoc3-core.patch`:
- `ext3-mballoc3-sles10.patch`:

- `ext3-nlinks-2.6.9.patch`: allows more than 32,000 subdirectories to be created in a given directory. This is achieved by setting `i_nlink` to 1 if overflowed (if number of subdirectories is greater than 32,000, addressed by a 15 bit `i_nlink` counter) (also used in the Linux ext4 filesystem).

- `ext3-ialloc-2.6.patch`: changes the inode allocation policy for OSTs to avoid full groups (i.e., to skip full groups). There is a bitmap at the beginning of each inode group to show which group is fully used, which group is partially used, and which group is free. The superblock contains information about how many inodes are free in different inode groups. This patch gives information about an inode group, whether there are any free inodes in that group or not, without making a full inode group scan.

- `ext3-disable-write-bar-by-default-2.6-sles10.patch`: turns off write barriers in a journal which cause cache flushes.

- `ext3-uninit-2.6-sles10.patch`: provides identifying uninitialized block groups for increasing the e2fsck performance (also used in the Linux ext4 filesystem).

- `ext3-nanosecond-2.6-sles10.patch`: provides nanosecond resolution timestamps for inodes (also used in the Linux ext4 filesystem).

- `ext3-inode-version-2.6-sles10.patch`: provides inode version on disk for version based recovery (also used in the Linux ext4 filesystem).

- `ext3-mmp-2.6-sles10.patch`: provides multi-mount protection to avoid double mount in Linux High-Avaliabilty (HA) environment.

- `ext3-fiemap-2.6-sles10.patch`: provides file extent mapping API (efficient fragmentation reporting) (also used in the Linux ext4 filesystem).

- `ext3-statfs-2.6-sles10.patch`: provides `statfs` speedup by just gathering info from superblock without diving deeper (also used in Linux ext4 filesystem).

- `ext3-block-bitmap-validation-2.6-sles10.patch`: verifies bitmaps on disk are sane to avoid cascading corruption (also used in the Linux ext4 filesystem).

- `ext3-get-raid-stripe-from-sb.patch`: stores RAID layout in ext3 superblock to optimize allocation (used by allocator and the information is written by mkfs) (also used in the Linux ext4 filesystem).

12 Future Work

Lustre was initiated and funded, almost a decade ago, by the U.S. Department of Energy (DOE) Office of Science and National Nuclear Security Administration (NNSA)

Laboratories to address the need for an open source, highly scalable, high-performance parallel filesystem on then-present and future supercomputing platforms. Throughout the last decade, while satisfying the scalability and performance requirements of the various supercomputing platforms deployed not only by DOE Laboratories but also by other domestic and international industry and research institutes, Lustre has become increasingly large and complex. This report only scratches the surface of the current 1.6 version of the Lustre source code base. Because Lustre is a moving target, keeping documentation up to date to address day-to-day user problems and to help meet future requirements will require a substantial effort. Limited by time and resources the authors did not include the following topics that merit documentation:

- This documentation is based on Lustre code base `b1.6` as the reference implementation. However, significant changes occurred on the `b1.8` branch and the upcoming 2.0 release. Any future effort should take this into consideration.
- Failure recovery has been a constant theme for many bug fixes and new Lustre feature development. It also has a direct bearing on providing insight on the Lustre diagnosis we do on a daily basis. Though we touch on it in this report, it would be desirable to provide a holistic view on the subject from the Lustre kernel support perspective.
- Lustre scalability is another topic in which we have a vested interest. A centralized discussion on its kernel support, status, limits, and recent bug fixes and feature enhancement would be highly beneficial.
- Quota support was also left out of this report. A detailed analysis of this subject in future updates will be benefical for both the Lustre developers and the user community.
- Security (e.g., Kerberos, in terms of computer network authentication) will soon be a default requirement for Lustre. Coverage of this topic in future updates would provide a more complete picture of Lustre.

The authors have shared their insights and understanding of Lustre as it stands today. The documentation on Lustre filesystem internals may never be complete because of the ever-changing nature of the Lustre source code base. It is the authors' hope that the Lustre community will make a collective effort to continue this work.

Acknowledgments

The authors gratefully acknowledge the proof-reading suggestions and advices of David Dillow and James Simmons. We would also like to express our thanks to Priscilla Henson and Amy Harkey for their tireless and meticulous efforts in editing this report.