DNE2 High Level Design

Introduction

With the release of DNE Phase I Remote Directories Lustre* file systems now supports more than one MDT. This feature has some limitations:

1. Only an administrator can create or unlink a remote directory. Create and unlink are the only 'cross-MDT' operations to be allowed in Phase I. All other cross-MDT operations will return EXDEV.
2. Cross MDT operations must be synchronous and metadata performance may be impacted especially when DNE is based on ZFS.
3. Moving files or directories between MDTs can only be achieved using copy and remove commands. As a result, data objects on OST will be moved resulting in redundant data transfer operations.
4. All of name entries of a directory must be on one MDT, so the single directory performance is the same as single MDT filesystem.

DNE Phase II will resolve these issues. This document is divided into two sections. The first section 'A-sync, Cross-MDT Operation Design' is concerned with resolution of the first three issues above. A separate section 'Striped Directories Design' describes a design to resolve the remaining issue.

A-sync, Cross-MDT Operation Design

From the first three limitations enumerated in the introduction, the most important one is how to implement asynchronous cross-MDT updates and its recovery. An assumption is made that the file system may become inconsistent after the recovery in some rare cases, and both servers and clients should be able to detect such inconsistency and return proper error code to the user. In the mean time, LFSCK will be able to detect such inconsistency and attempt to resolve them. This design document assumes knowledge of the DNE phase II Solution Architecture, DNE Phase 1 Solution Architecture and DNE Phase 1 High Level Design.

Definitions

Operation and Update

Operation means one complete metadata operation i.e open/create a file, mkdir or rename. A request from a client usually includes only one metadata operation. The MDT will decompose the operation into several **updates**, for example mkdir will be decomposed into name entry insertion and object create.

Master and slave(remote) MDT

In DNE, client typically sends the metadata request to one MDT called **master MDT** for this request. The master MDT then decomposes the operation into updates, and redistributes these updates to other MDTs, which are called **slave MDT** or **remote MDT** for this request.

Functional Statements

In DNE Phase II

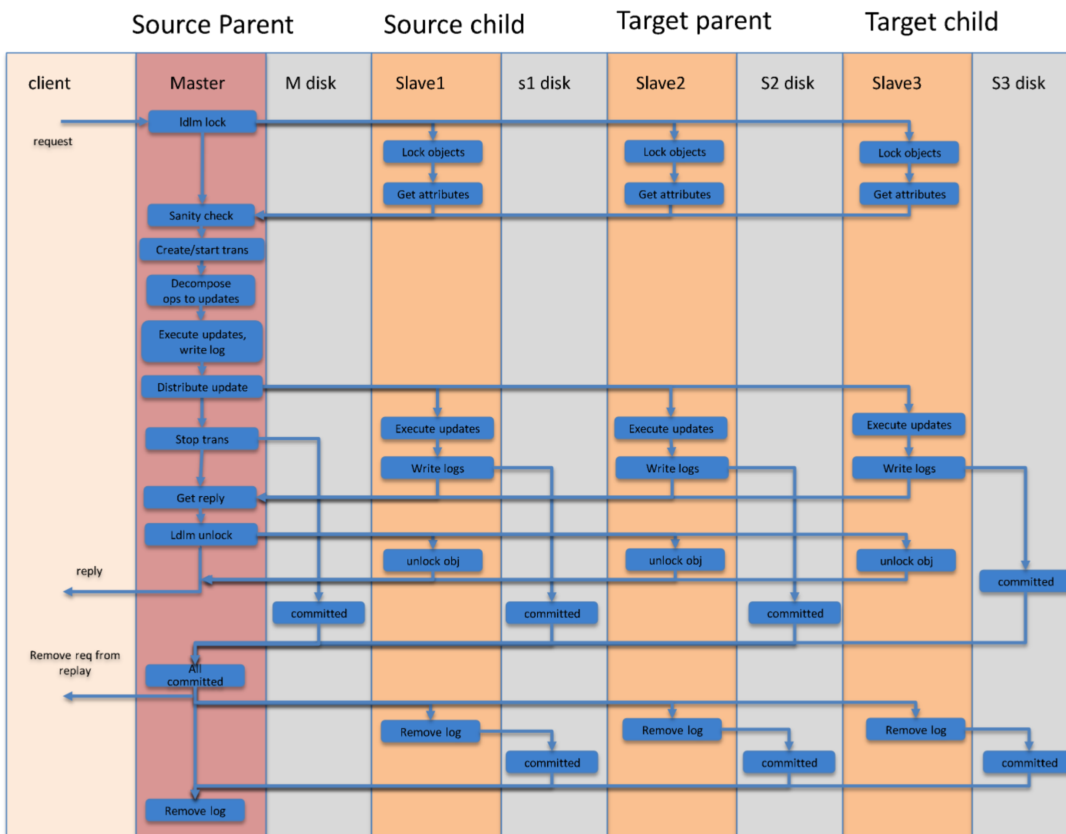
1. All metadata operations are allowed to be cross-MDT, which will be asynchronous.
2. Normal users (without administrator privilege) can perform cross-MDT operations.
3. Migration tool will be provided to move individual inodes from one MDT to another MDT, without introducing redundant data object transfers on OSTs.

Implementation

In DNE Phase I, the master MDT collects the updates in the transaction declare phase, and then sends these updates to other MDTs during transaction start. For local transactions the declare phase is only for reserving the resource, like journal credit etc. To unify the transaction phase for local and remote operation DNE Phase II will collect updates in the execution phase, i.e. between transaction start and stop, then distribute updates at transaction stop. The process is:

1. The client sends the request to the master MDT.

2. The master MDT enqueues all LDLM locks and get back the object attributes, then cache those attributes on the master MDT, and do sanity check on the master MDT.
3. The master MDT creates and starts the transaction, and decomposes the operation into updates in MDD layer, which might include both local and remote updates.
4. The master MDT executes its local updates during the transaction proper.
5. In transaction stop, the master MDT will first generate the transno = `ur_batchid` for the operation, and generate an additional update to the master's `last_rcvd` file containing the `lsd_client_data` (master transno, client XID, object pre_versions) for that operation, and add this to the update log. If recovery of the master MDT is needed, it will update `last_rcvd` by executing the update log.
6. The master MDT distributes the full update to all of remote MDTs, using the RPC XID = transno = `ur_batchid`.
7. All of MDTs will execute their local updates asynchronously, and also write all of the updates into its local log, then reply with their transno to the master MDT.
8. After master MDT gets all replies from slave MDTs, it releases the LDLM locks and replies to the client with the master transno generated in 5, and client will add the request into the replay list.
9. After the operation and its update log are committed to disk on master MDT, it will piggyback the last_committed_transno to the client, and client will remove the request from the replay list.
10. When those updates are committed on the slave MDT(s), they will notify the master MDT using normal `last_committed_transno` in RPC replies or pings.
11. After the master MDT sees all of the remote updates are committed on the slave MDTs, it will flag the first update for that `ur_batchid` in its local update record as globally committed, so that it knows this record does not need to be replayed.
12. When the local update record has committed, the master MDT send requests to all of remote MDTs to cancel their corresponding update records (identified by `ur_master_index` + `ur_batchid`). The remote MDTs will use the `ur_master_index` + `ur_batchid` to be able to cancel all of their update records belonging to that operation. These may be in memory (e.g. hash table) for easier location, but in case of a crash the update llog processing will load all of the existing updates and order them by `ur_master_index` + `ur_batchid` for later processing.
13. After the cancellation of remote update llog records is committed to disk, remote MDT will notify the master MDT (through normal `last_committed`) and the master MDT will cancel its local update llog record. If the master MDT crashes before the local update llog has been cancelled, it will know not to replay this operation to the slave MDTs because the record will be marked as globally committed.



Note: ldlm lock does not need during recovery.

1. If the replay request comes from client, the master MDT will re-enqueue the lock for the replay request.
2. The failover MDT will not accept new request from clients during recovery, besides commit on share(COS) will be applied for all cross-MDT operation, which make sure all of the conflict updates has been committed to disk, so any cross-MDT replay updates should not be conflicted, i.e. no need ldlm lock for replay between MDTs neither.

Update request format

As described earlier, one metadata operation will be decomposed into several updates. These updates will be distributed to all other MDTs by update RPC. In DNE Phase II each RPC only includes updates for single operation. The format for update RPC is:

```
enum update_rec_flags {
        OUT_UPDATE_FLAG_TYPE      = 0x000000001, /* 0 = MDT, 1 = OST */
        OUT_UPDATE_FLAG_SYNC      = 0x000000002, /* commit update before reply */
        OUT_UPDATE_FLAG_COMMITTED = 0x000000004, /* ur_batchid is committed globally
*/
        OUT_UPDATE_FLAG_NOLOG     = 0x000000008, /* idempotent update does not need to
be logged */
};

struct update_rec_v2 {
        __u16         ur_update_type;             /* OUT_* update type */
        __u16         ur_master_index;            /* master MDT/OST index */
        __u32         ur_flags;                   /* master target type, globally
committed, sync, log?, etc */
        __u64         ur_batchid;                 /* transno of master MDT in operation
*/
        struct lu_fid ur_fid;                     /* FID operation applies to */
        __u32         ur_lens[UPDATE_BUF_COUNT]; /* length of each update buffer,
rounded up to 8 bytes */
        char          ur_bufs[0];                 /* per operation data values,
multiple of 8 bytes */
};

struct update_header {
        __u32 uh_magic;                           /* UPDATE_BUFFER_MAGIC_V2 */
        __u32 uh_count;                           /* number of update records in
request */
        __u32 uh_bufs[uh_count];                  /* length of each struct
update_rec_v2 */
};
```

Open/create regular remote files

When create regular remote files,

1. Client allocates the fid and sends create request to the master MDT where the file is located.
2. Master MDT creates the regular file by the fid.
3. Slave MDT inserts the name entry into its parent.

Note, open/create are not allowed for remote regular file in this phase, So if the user want to open/create a regular file, he has to create the regular file first, then open the file in the separate system call.

Unlink remote files/directories

In common with DNE Phase I, DNE Phase II clients will send unlink request to the MDT where the inode is located:

1. The client sends unlink request to the master MDT.

2. The master MDT enqueue the LDLM lock of the remote parent and the file.
3. The master MDT then decrease the nlink of the inode, if the nlink is zero,
    a. if the file is being opened, move the file to ORPHAN directory.
    b. if the file is not being opened, destroy the file.
4. The remote MDT will remove the name entry from the parent.

Rename remote files/directories

In contrast to other cross-MDT metadata operations, rename between multiple MDTs involves four objects, which might be in different MDTs. This adds additional complexity. Additional care must be taken before rename: the relationship between the source and the target must be checked to avoid moving the parent into the subdirectory of its child. The checking process is protected by a single global lock to ensure the relationship will not change during the check. The global lock will be put into MDT0. Since this global lock is only used to protect the checking process, and the relationship can not be changed after holding all of LDLM locks, the global rename lock can be dropped after acquiring all of LDLM locks.

(rename dir_S/src dir_T/tgt MDT1(master MDT) holds dir_S MDT2 holds src, MDT3 holds dir_T, MDT4 holds tgt)

1. The client sends rename request to MDT4 if the `tgt` object exists, otherwise to MDT2 where the `src` object exists (though this is not a hard requirement). This is the master MDT.
2. If the clients sends the RPC to an MDT and it looks up the `tgt` name under DLM lock and `tgt` object exists on a remote MDT, the MDT will return `-EREMOTE` and the client must resend the RPC to the MDT with the `tgt` object.
3. If the renamed object is a directory, the master MDT acquires the global rename lock. The master MDT gets the LDLM lock of `dir_S` and `dir_T` according to their FID order, then gets the LDLM lock of their child name hashes.
4. If the renamed object is a directory the master MDT checks the relationship between the `dir_S` and `dir_T`. If the `dir_S` is the parent of `tgt`, the rename is not allowed
5. MDT1 deletes entry `src` and set ctime/mtime of `dir_S`.
6. If the renamed object is a directory MDT2 deletes old `dir_S` "`..`" entry and insert new `dir_T` "`..`" entry, sets ctime/mtime of `src` and also updates the `linkEA` of `src`.
7. The master MDT deletes old entry `tgt` if it exists, and insert new entry `tgt` with the `src` object FID, and also updates the link count of dir_T if this is a directory.
8. If the renamed object is a directory then the master MDT releases global rename lock
9. If `tgt` object exist, MDT4 destroys `tgt`.

All of these updates will be stored in the update logs on every MDT. If any MDT fails and restarts, it will notify other MDTs to send all these updates to the failover MDT, which then will be redo on this MDT, which will discussed in detail in failover section.

Migration

In DNE Phase II, migration tool (lfs mv file -i target_MDT) will be provided to help users to move individual inode from one MDT to another MDT, without moving data on OST.

Migrating regular files will be performed as follows:

(lfs mv file1 -i MDT3, MDT1 holds the name entry of file1, MDT2 holds file1, MDT3 will be the target MDT where the file will be migrated)

1. The client sends the migrate request to MDT1.
2. MDT1 checks whether the file is being opened, and return -EBUSY if it is opened by other process.
3. MDT1 acquires LAYOUT, UPDATE and LOOKUP lock of the file.
4. MDT3 create an new file with the same layout, and update linkEA.
5. MDT1 update the entry with new FID.
6. MDT2 destroy the old object, but if there are multiple links for the old object, it also needs to walk through all of the name-entries and update the FID in all these name entries.
7. MDT1 release LAYOUT, UPDATE and LOOKUP lock.
8. Client clears the inode cache of file1.

Migrating directories is more complicated:

((lfs mv dir1 -i MDT3, MDT1 holds the name entry of dir1, MDT2 holds dir1, MDT3 will be the target MDT where the file will be migrated)

1. The client sends the migrate request to MDT1, where the directory is located.
2. MDT1 checks whether the file is being opened, and return -EBUSY if it is opened by other process.
3. MDT1 acquires LAYOUT, UPDATE and LOOKUP lock of the file.
4. MDT3 create the new directory. Note: if it is non-empty directory, MDT1 needs to iterate all of entries of the directory, and send them to MDT3, which will insert all of the entries on the new directory, and also the linkEA of each children needs to be updated.

5. MDT2 destroy the old directory.
6. MDT1 update the entry with new FID.
7. MDT1 release layout, UPDATE and LOOKUP lock.

When the entire directory is being migrated from one MDT to a second MDT, individual files and directories will be migrated from the top to bottom, i.e. the parent will be migrated the new MDT first, then its children. By this way, if other process create the file/directories during the migration,

1. If the parent of the creation has been moved to the new MDT, the file/directory will be created on the new MDT.
2. If the parent of the creation has not been moved to the new MDT yet, the new created file/directory will be moved to the new MDT in the following migration.

This design ensures the all directories will be migrated to the new MDT in all cases.

After migrating the directory to the new MDT the directory on the old MDT will become an orphan, i.e. it can not be accessed from the namespace. The orphan can not be destroyed until all of its children are moved to the new MDT. In this way, migrating a directory does not need to update the parent FID in the linkEA of all of the children since all of children can still find its parent on the old MDT using fid2path during migration.

Failover

In DNE Phase I all of cross-MDT requests are synchronous and there are no replay requests between MDTs. This design simplifies ecovery between MDTs in DNE Phase I. With DNE Phase II, all of cross-MDT operations are asynchronous and there will be replay requests between MDTs. This makes recovery more complex than for DNE Phase I.

As described earlier the cross-MDT updates will be recorded on every MDTs. During recovery the updates will be sent to the failover MDT and are replayed there. With the exception of updates of the operation, the update to update the last rcvd on the master MDT will also be added in the update log. A new index method to update index(index_update) is used. This record will include:

1. Master MDT index so the operation will only be replayed by the master MDT during recovery.
2. local FID `{ FID_SEQ_LOCAL_FILE, LAST_RECV_OID, 0 }`, to present the `last_rcvd` file.
3. The lsd_client_data structure, the client UUID (to be used for the index key), and the rest of the body is the value.

During update, the master MDT will first locate the `last_rcvd` by FID, then locate record in the file by client UUID, then update the whole body of `lsd_client_data`.

Recovery in DNE Phase II is divided into three steps:

1. When one MDT restarts after a crash it will process all of the records in it's local update llog. It will batch up all of the updates with the same `ur_master_index, ur_batchid` and sends them in an `OUT_UPDATE` RPC to each of the remote targets that were part of the operation. Simultaneously, all other MDTs will be notified and they will check their own local update log, and all of the related records will be sent to the failover MDT.
    a. The MDT that receives the updates from other MDT will check whether the corresponding updates are already recorded in their local update llog.
        i. If the update was already committed, then the MDT will reply with an arbitrary `pb_transno` < `pb_last_committed`.
        ii. If the updates do not exist in the update llog they will compare the master transno in the update record with the transno in the last_rcvd, if the transno in update record is smaller than the one in the last_rcvd, it means the master already sent the update to this MDT, and the update is already being exected and committed, and the update log has been deleted, so it will also return an arbitrary smaller transno as above. If the transno in the update record is larger, it will replay the update with a new transno.
        iii. In all of cases, the MDT will reply to the sender with the transno.
    b. If the sender is the recovering MDT, which is the master for this operation, it will build the in-memory operation state to track the remote updates, and when all of the remote updates have committed, it can cancel the local update record.
2. Then client will send replay/resend request to the failover MDT,
    a. The master MDT will check if the request exists in the update log by the request xid.
        i. If it does not exist, it will compare the request transno with its own transno, only replay the request if its transno is bigger than the last transno (lcd_last_transno) of this MDT.
        ii. If it does exist, it means the recovery between MDTs has been handled in this case. An arbitrary smaller transno will be returned and the client will remove the request from the replay list.
3. If there are any failures during the above 2 steps, lfsck daemon will be triggered to fix the filesystem.

Recovery of cross-MDT operations requires the participation of all involved MDTs. In case of multiple MDT failures, normal service cannot therefore resume until all failed MDTs fail over or reboot. The system administrator may disable a permantently failed MDT(by lctl deactivate) to

allow recovery to complete on the remaining MDTs.

Failure cases

Failures	Failover
Both master and slave fail and updates have been committed on both MDTs.	The master will replay the update to the slave, and the slave will know whether the update has been executed by checking the update llog and generate the reply.
Both master and slave fail and updates have not been committed on both MDT yet.	Client will do normal replay (or resend if no reply), master will redo whole operation from scratch. If the client has also failed, nothing left to be done.
Both master and slave fails and updates have been committed on master MDT but not on slave MDT.	The master resends the update to slaves and slaves will redo updates; Client may resend or replay, and this will be handled by client `last_rcvd` on master.
The master is alive and the slave fails without committing the update.	The master replays (or resend if no reply) updates to the slave MDT, and the slave will redo updates.
The master is alive and the slave fails having committed the update.	The master replays (or resend if no reply), the slave will generate reply from the master `last_rcvd` slot based on XID (== master transno, if resend)
The master fails without commit the slave is alive.	The slave replays updates to the master MDT when it restarts. The master will check whether the update has been executed by checking its local update llog, and redo the update if not found. This also updates the client's `last_rcvd` slot from the update, so if the client replays or resends it can be handled normally.

Commit on Share

During recovery, if one update replay fails all related updates may also fail in the subsequent replay process. For example, client1 creates a remote directory on MDT1 and its name entry is on MDT0: other clients will create files under the remote directory on MDT1. If MDT0 fails and the name entry insertion has not yet been committed to disk. If the recovery fails for some reason, i.e. the directory is not being connected to the name space at all, all of the files under this directory will not be able to be accessed. To avoid this, commit on share will be applied to cross-MDT operation. i.e. If the MDT finds the object being updated was modified by some previous cross-MDT operation, this cross-MDT operation will be committed first. So in the previous example before creating any files under remote directory the creation of the remote directory must be committed to disk first.

Commit on Share (COS) will be implemented by COS lock based on the current local COS implementation. During cross-MDT operation, all locks of remote objects(remote locks) will be hold on the master MDT, and all of remote locks will be COS lock. If these COS locks are being revoked, the master MDT will not only sync itself, but also sync the remote MDTs.

For example, these two consecutive operations: 1. mv dir_S/s dir_T/t 2. touch dir_T/s/tmp (MDT1 holds dir_S, MDT2 holds s, MDT3 holds dir_T, MDT4 holds t)

1. Client sends rename request to MDT1
2. MDT1 detects remote rename and holds LDLM COS locks for all four objects, and finish rename, and four LDLM locks are cached on MDT1.
3. Client sends open/create request to MDT2
4. MDT2 enqueue LDLM lock for s, MDT1 revoke the lock of s, because it is a COS lock, it will do sync on all of MDTs involve in the previous rename, i.e. MDT1, MDT2, MDT3, MDT4.

Compatibility

MDT-MDT

In DNE Phase I updates between MDTs are synchronous. In DNE Phase II updates are asynchronous. To avoid complications introduces with multiple different MDT versions, DNE Phase II requires all MDTs have to be of the same version, i.e. they must be upgraded at the same time.

MDTs will check versions during connection setup and deny the connect requests from old MDT version.

MDT-OST

There are no protocol changes between MDT and OST in DNE Phase II.

CLIENT-MDT

In DNE Phase II rename request will be sent to the MDT where the target file is located. This is different from DNE Phase I. An old client (<= Lustre software version 2.4.0) will still send the request to the MDT where the source parent is, and the source parent will return `-EREMOTE` to the old client. A 2.4.0 client does not understand `-EREMOTE so` a patch will be added to 2.4 series to redirect rename request to the MDT where the target file is, if it gets `-EREMOTE` from the MDT.

Disk-compatiblity

The striped directory will be introduced in DNE Phase II so a compatible flag in the LMA of the stripe directory will be added. If an old MDT (< Lustre software version 2.5) tries to access the striped directory, it will get `-ENOSUPP` error.

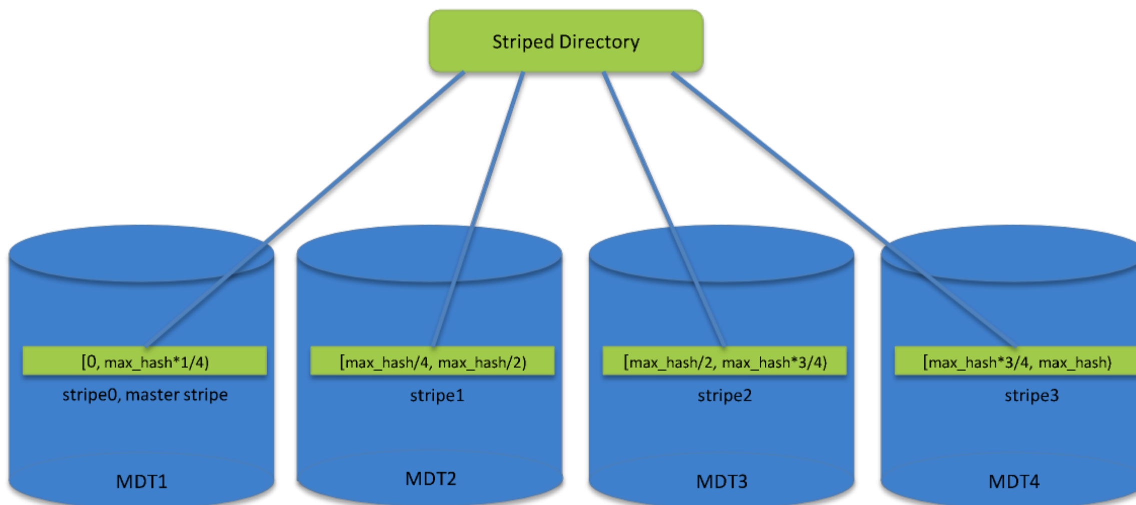
Striped Directories Design

Introduction

In DNE Phase I all of name entries of one directory will be only in a single MDT. As a result, single directory performance is expected to be the same as single MDT file system. In DNE Phase II a striped directory will be introduced to improve the single directory performance. This document will discuss how striped directory will be implemented. It assumes the knowledge of DNE phase II async cross-MDT operation High Level Design and DNE phase I Remote Directory High Level Design.

Functional Statement

Similar to file striping, a striped directory will split the name entries across multiple MDTs. Each MDT keeps directory entries for certain range of hash space. For example, there are N MDTs and hash range is 0 to `MAX_HASH`, first MDT will keep records with hashes `[0, MAX_HASH/N - 1]` , second one with hashes `[MAX_HASH / N, 2 * MAX_HASH / N]` and so on. During file creation, LMV will calculate the hash value by the name, then create the file in the corresponding stripe on one MDT. It will also allow the user to choose different hash function to stripe the directory. The directory can only be striped during creation and can not be re-striped after creation in DNE phase II.



Definition

The first stripe of each striped directory will be called **master stripe**, which is usually in the same MDT with its parent. Other stripes will be called **r**

emote stripes.

Logical Statement

Similar to a striped file, a client will get directory layout information after lookup and then build the layout information for this directory in LMV. For any operation under the striped directory, the client will first calculate the hash value by name then get the stripe by hash and layout. Finally, the client will send the request to the MDT where the stripe is. If a large number of threads access the striped directory simultaneously, each thread can go to different MDTs and these requests can be handled by each MDT concurrently and independently. The single directory performance will be improved by this way.

Directory Layout

The directory layout information will be stored in the EA of every stripe as follows:

```
struct lmv_mds_md {
        __u32 lmv_magic;                        /* stripe format version */
        __u32 lmv_count;                        /* stripe count */
        __u32 lmv_master;                       /* master MDT index */
        __u32 lmv_hash_type;                    /* dir stripe policy, i.e. indicate
which hash function to be used*/
        __u32 lmv_layout_version;               /* Used for directory restriping */
        __u32 lmv_padding1;
        __u32 lmv_padding2;
        __u32 lmv_padding3;
        char lmv_pool_name[LOV_MAXPOOLNAME];   /* pool name */
        struct lu_fid lmv_data[0];              /* FIDs for each stripe */
};
```

`lmv_hash_type` indicates which hash function the directory will use to split its name entries.

Directory stripe lock

Currently all of name entries of one directory are protected by the UPDATE lock of this directory. As a result, the client will invalidate all entries in this directory during Update lock revocation. In striped directory each stripe has its own UPDATE lock and if any threads try to modify the stripe directory the MDT only needs acquire the single stripe UPDATE lock. Consequently, the client will only invalidate name entries of this stripe, instead of all of entries of the directory. When deleting the striped directory the MDT needs to acquire each of the stripe locks; When performing readdir of the striped directory, the client must to acquire each stripe lock to cache the directory contents. Stripe locks do not need to be acquired simultaneously.

Create striped directory

Creating a striped directory is similar to creating a striped file:

1. The client allocates FIDs for all stripes and sends the create request to the master MDT.
2. The master MDT sends object create updates to each remote MDT to create the stripes.
3. For each remote stripe, the parent FID in LinkEA will be the Master stripe FID, which will also be put into the ".." directory of each remote stripe, i.e. the remote stripes will physically be remote subdirectories of the master stripe to satisfy lfsck. During readdir, LMV will ignore this subdirectory relationship, and recognize it as individual stripe of the directory (it will be collapsed by LMV on the client with the layout and skipped during readdir.) This design simplifies LFSCK consistency checking and reduces the number of objects modified during rename (for ".." and LinkEA).

Delete striped directory

Client sends delete requests to the Master MDT, then Master MDT acquires all of stripe locks of the directory. The Master MDT checks if all of stripes are empty and then destroys all of the stripes.

Create/lookup files/directories under striped directory

When a file/directory is being created/looked up under stripe directory:

1. Client will first calculate the hash according to the name and `lmv_hash_type` of the striped directory. Next, the client gets the MDT index according to the hash and sends the create/lookup request to that MDT.
2. MDT will create/lookup the file and directories independently. Note: when creating the new directory, MDT only needs to modify the attributes of the local stripe, like increase nlink, mtime, so to avoid sending attrset updates between MDT. It also means when client tries to retrieve the attribute of striped directories, it needs to walk through all of stripes on different MDT, then merge attributes from each stripe.

Readdir of striped directory

During readdir() a client will iterate over all stripes and for each stripe it will get a stripe lock and then read directory entries. Each directory's hash range should be in the range $[1..2^{63}-1]$. The readdir() operation will proceed in hash order concurrently among all of the stripes that make up the directory, and the client will perform a merge sort of the hashes of all the returned entries to a single stream, up to the lowest hash value at the end of the returned directory pages. This allows a single 64-bit cookie to represent the readdir offset within all of the stripes in the directory. There is no more chance of hash collision with the readdir cookie in a striped directory than there is with a single directory of equivalent size.

Getattr of striped directory

Client iterate over all of the stripes to get attributes from all stripes and then merge them together.

1. size/blocks/nlink: add all together from every stripe.
2. ctime/mtime/atime: choose the newest one as the xtime of the striped directory.
3. uid/gid: should be same for all stripes.

Rename in the same striped directory

Client sends the rename request to the MDT where the master stripe of the source parent is located. If rename is in the same stripe it is the same as rename in the same directory. If the rename is under the same striped directory but between different stripes on different MDTs:

(`mv dir_S/src dir_S/tgt`, `dir_S` is striped directory, MDT0 holds the master stripe, MDT1 holds `src`, MDT2 holds `tgt`).

1. Client sends the rename request to MDT2.
2. MDT2 acquires the LDLM locks (both inode bits and hash of the file name) of the source and target stripe according to their FID order.
3. MDT1 deletes entry `src`, sets mtime of the stripe, updates `linkEA` of `src`, and if `src` is directory, decreases the nlink of the local stripe.
4. MDT2 deletes entry `tgt`, inserts entry `src`, and if `tgt` is directory, increases the nlink of the local stripe,

Rename between different striped directory

Rename between different striped directories is a more complicated case with potentially six MDTs involve in the process:

(`mv dir_S/src dir_T/tgt`, MDT1 holds the source stripe of `dir_S` where the name entry of `src` is located, MDT2 holds `src` object, MDT3 holds the target stripe of `dir_T` where the name entry of `tgt` is located, MDT4 holds `tgt` object)

1. The client sends rename request to MDT4 if the `tgt` object exists, otherwise to MDT2 where the `src` object exists (though this is not a hard requirement). This is the master MDT.
2. If the clients sends the RPC to an MDT and it looks up the `tgt` name under DLM lock and `tgt` object exists on a remote MDT, the MDT will return `-EREMOTE` and the client must resend the RPC to the MDT with the `tgt` object.
3. If the renamed object is a directory, the master MDT acquires the global rename lock. The master MDT gets the LDLM lock of `dir_S` and `dir_T` stripe according to their FID order, then gets the LDLM lock of their child name hashes.
4. If the renamed object is a directory the master MDT checks the relationship between the `dir_S` and `dir_T` stripes. If the `dir_S` is the parent of `tgt`, the rename is not allowed
5. MDT1 deletes entry `src` and set ctime/mtime of `dir_S`.
6. If the renamed object is a directory MDT2 deletes old `dir_S` "`..`" entry and insert new `dir_T` "`..`" entry, sets ctime/mtime of `src` and also updates the `linkEA` of `src`.
7. The master MDT deletes old entry `tgt` if it exists, and insert new entry `tgt` with the `src` object FID, and also updates the link count of local stripe if this is a directory.
8. If the renamed object is a directory then the master MDT releases global rename lock

9. If `tgt` object exist, MDT4 destroys `tgt`.

If the object being renamed is itself a striped directory, only the master stripe will have its "`..`" and `linkEA` entry updated.

LinkEA

`LinkEA` is used by `fid2path` to build the path by object FID. The LinkEA includes the parent FID and name. During `fid2path` a MDT will lookup the object parents to build the path until the root is reached. For a striped directory the master stripe FID will be stored into the `linkEA` of each other stripe. The parent FID of the striped directory will be put into the master stripe. A a result, if the object is under a striped directory the MDT will get stripe object first then locate the master stripe and then continue the `fid2path` process. For clients that do not understand striped directories (if supported), this may appear as a "`//`" component in the generated pathname, which is will fail safe.

Change log

An operation for a striped directory will be added to change log in the same way as a normal directory. The added operations include: create directory, unlink directory, create files under striped directory etc. Currently there are two users for change log,

1. `lustre_rsync` may be enhanced to understand striped directories:
   a. If `lustre_rsync` target is Lustre file system, it will try to recreate the stripe directory with original stripe count. If it succeeds, it will reproduce all operations under the striped directory.
   b. If it can not create the striped directory with the stripe count (for example there are not enough MDT on the target file system,) or the `lustre_rsync` target is not Lustre file system, it will create a normal directory, and all of striped directory operation will be converted to normal directory operation.
   c. Besides the original directory creation, all of the `lustre_rsync` operations proceed as normal.
2. The striped directory implementation does not interact with HSM. This behaviour is consistent with DNE Phase I.

Recovery

Recovery of striped directory will use the redo log as described in DNE phase II async cross-MDT operation High Level Design.

In case of on-disk corruption in a striped directory, the LFSCK Phase III MDT-MDT Consistency project will address the distributed verification and repair.

Compatibility

Since old clients (<= Lustre software version 2.4) do not understand striped directories -ENOSUPP will be returned when old clients try to access the striped EA on the new MDT (>= Lustre software version 2.6).

---

*Other names and brands may be the property of others.