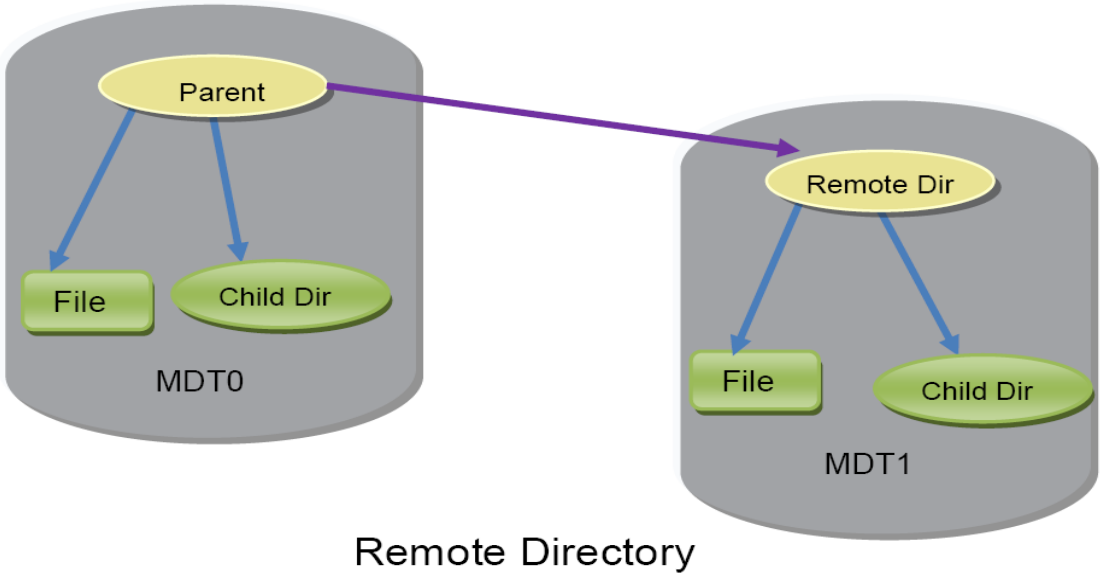
Remote Directories High Level Design

Introduction

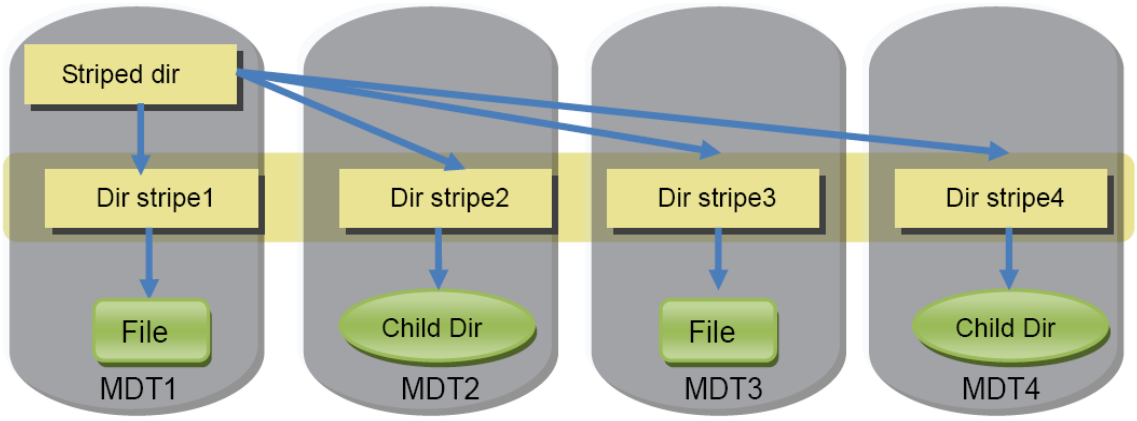
Distributed Namespace (DNE) allows the Lustre namespace to be divided across multiple metadata servers. This enables the size of the namespace and metadata throughput to be scaled with the number of servers. An administrator can allocate specific metadata resources for different sub-trees within the namespace.

This project is split into 2 phases

Phase 1
Remote Directories Lustre sub-directories are distributed over multiple metadata targets (MDTs). Sub-directory distribution is defined by an administrator using a Lustre-specific mkdir command.



Phase 2
Striped Directories The contents of a given directory are distributed over multiple MDTs.



Striped Directory

This document is concerned exclusively with the first phase: Remote Directories.

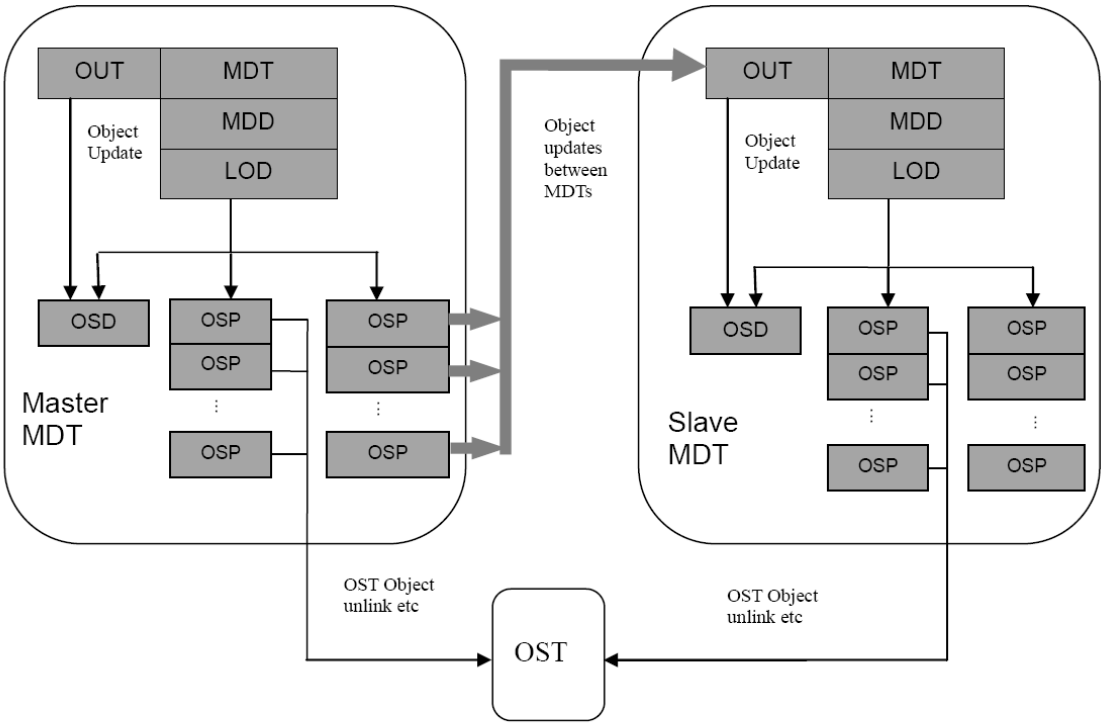
This document assumes knowledge of the DNE Phase 1 Solution Architecture the specifies all the requirements and presents the solution proposal.

Implementation

Architecture

DNE will be based on the Orion MDS stack. The Orion MDS stack is composed of the following layers:

- Metadata Target (MDT): unpacks requests and handles ldlm locks.
- Object Update Target (OUT): In the same layer as the MDT. Handles updates between MDTs. An update is a low-level modification to a storage object that applies directly to OSDs. For example, increasing link count. An operation is a complete and consistent modification to the file system that applies to MDD layer. for example, rename or unlink.
- Metadata Device (MDD): decomposes the received request(operation) into object updates.
- Logical Object Device (LOD): mapping layer manages the communication between targets: MDT to OST, and MDT to MDT (OUT).
- Object Storage Device (OSD): local object storage layer to handle local object updates.
- Object Synchronous Proxy (OSP): is a synchronous proxy for the remote OSD that is used to communicate with an other MDT/OST.



New MDS layer

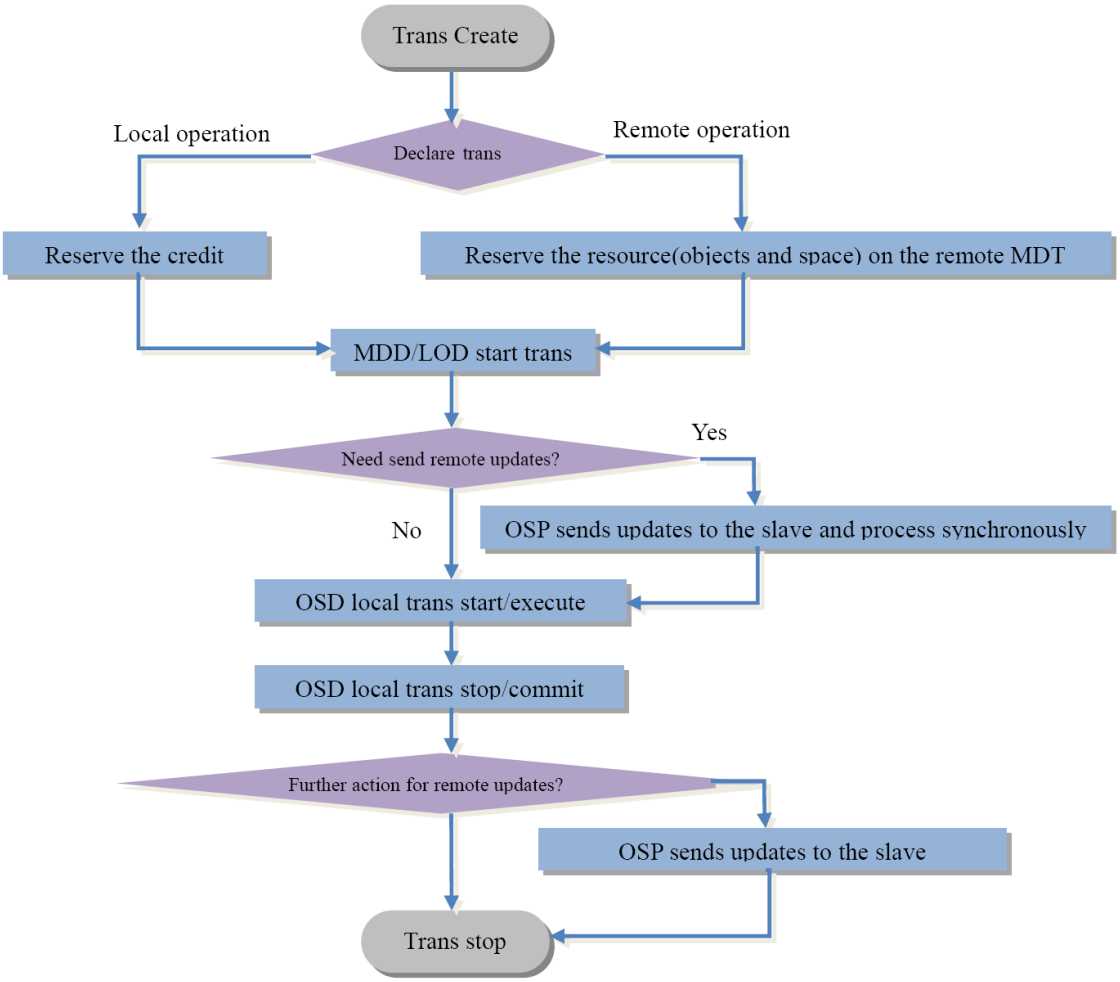
After the MDS receives a request from a client:

1. MDT unpacks the request and acquires ldlm locks for the operation.
2. MDD separates the operation into multiple object updates. For example the creation operation will be parsed into 'name entry insertion' and 'object creation'.
    - MDD/LOD does not know whether these updates are local or remote.
    - These updates are executed inside a transaction (see the following section for details.)
3. LOD checks if the updates are local or remote (i.e. executed on a remote MDT.)
    - Local updates will be sent to local OSD.
    - Remote updates will be sent to the OSP. The OSP will send updates to the remote MDT. To simplify recovery in the face of common failures, these updates will be processed synchronously on remote MDT. Only object updates are transferred between MDTs.

A transaction consists of four stages:

1. Transaction Create: create the transaction. The updates of this operation can be attached to the transaction at this stage.
2. Transaction Declare: reserve the resource for the transaction to guarantee the updates will succeed in the following execution phase.
    - For local updates, transaction credits are reserved. For remote updates, objects or spaces on the remote MDT are reserved. Note: If an error occurs during the declare phase, resource are released and an error is returned.
    - There is an execute and undo callback for each update.
3. Transaction Start: start the transaction, which is the update execute phase. Execute callbacks are called.
    - For remote updates, if a remote update is called before local update, (for example creating remote directory,) the execute callback will send RPC to the remote MDT to process the update synchronously.
    - If there any errors occur during this process the correspondent undo callback will be called to cancel the update.
4. Transaction Stop: the callback of each update is executed to end the transaction. Note: for unlink remote

directory, the remote directory will be destroyed after the local update has been committed to disk. Remove Remote Directory section below has more detail.



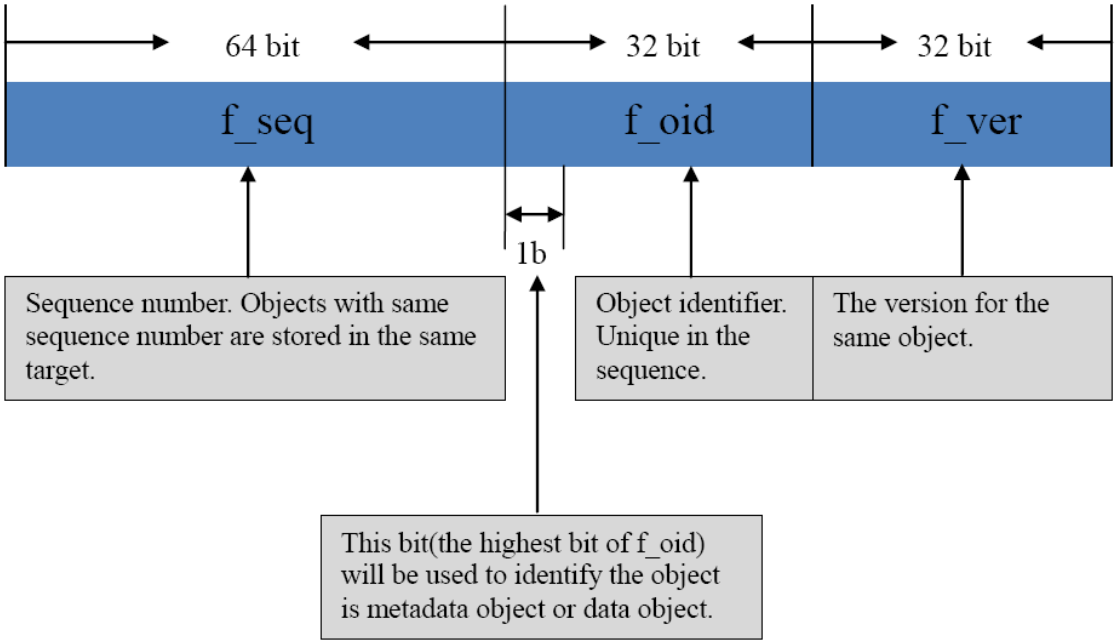
For DNE Phase 1, only create and unlink of the remote directory is supported. For this reason, only create and unlink are included in the the OSP API below. API calls are paired to provide synchronous behavior.

OSP OBJECT API	Description
`osp_md_declare_object_create` `osp_md_object_create`	Create an object remotely on another MDT.
`osp_md_declare_ref_add` `osp_md_object_ref_add`	Increase the refcount of a remote object.
`osp_md_declare_object_ref_del` `osp_md_object_ref_del`	Decrease the refcount of a remote object.
`osp_md_declare_attr_set` `osp_md_attr_set`	Set attribute of a remote object.

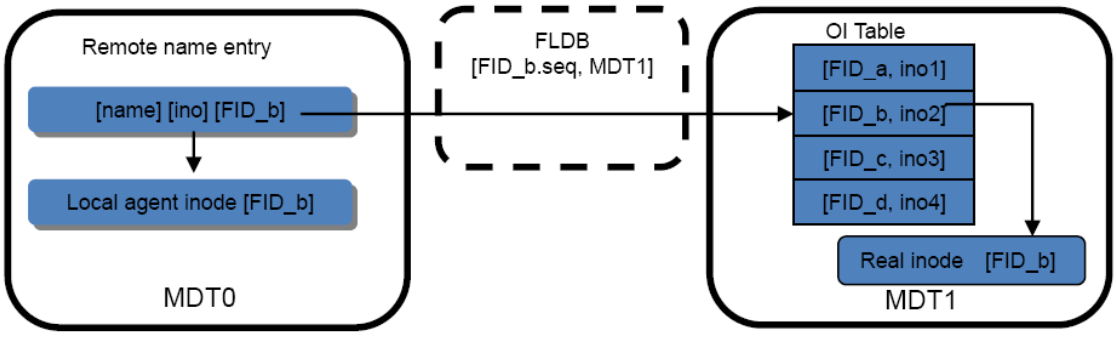
`osp_md_declare_xattr_set` `osp_md_xattr_set`	Set xattr of a remote object.
`osp_md_xattr_get` `osp_md_attr_get`	Get attr/xattr from a remote object.

Remote Directory

Lustre 2.1 introduced the File ID (FID). The FID uniquely identifies a file or a directory across the entire Lustre filesystem. A FID is a three-field data structure, illustrated in the figure below.



A FID is stored in two places: directory entry and the inode extended attribute (EA). FID in EA will be used by online checking tool (Inode Iterator and OI Scrub Solution Architecture). During lookup, the name entry will be located and the FID will be returned. The FID location database (FLDB) is then interrogated to provide the MDT index: this identifies the MDT on which the object resides. Finally, the object is identified by looking up the object index table on the appropriate MDT. In the case of a remote directory, FID will identify an object residing on another MDT. Every remote entry will have a local agent inode and the FID will be stored in the EA of this local inode. With this design, the FID of remote directory can be stored and retrieved after the filesystem is restored from backup.



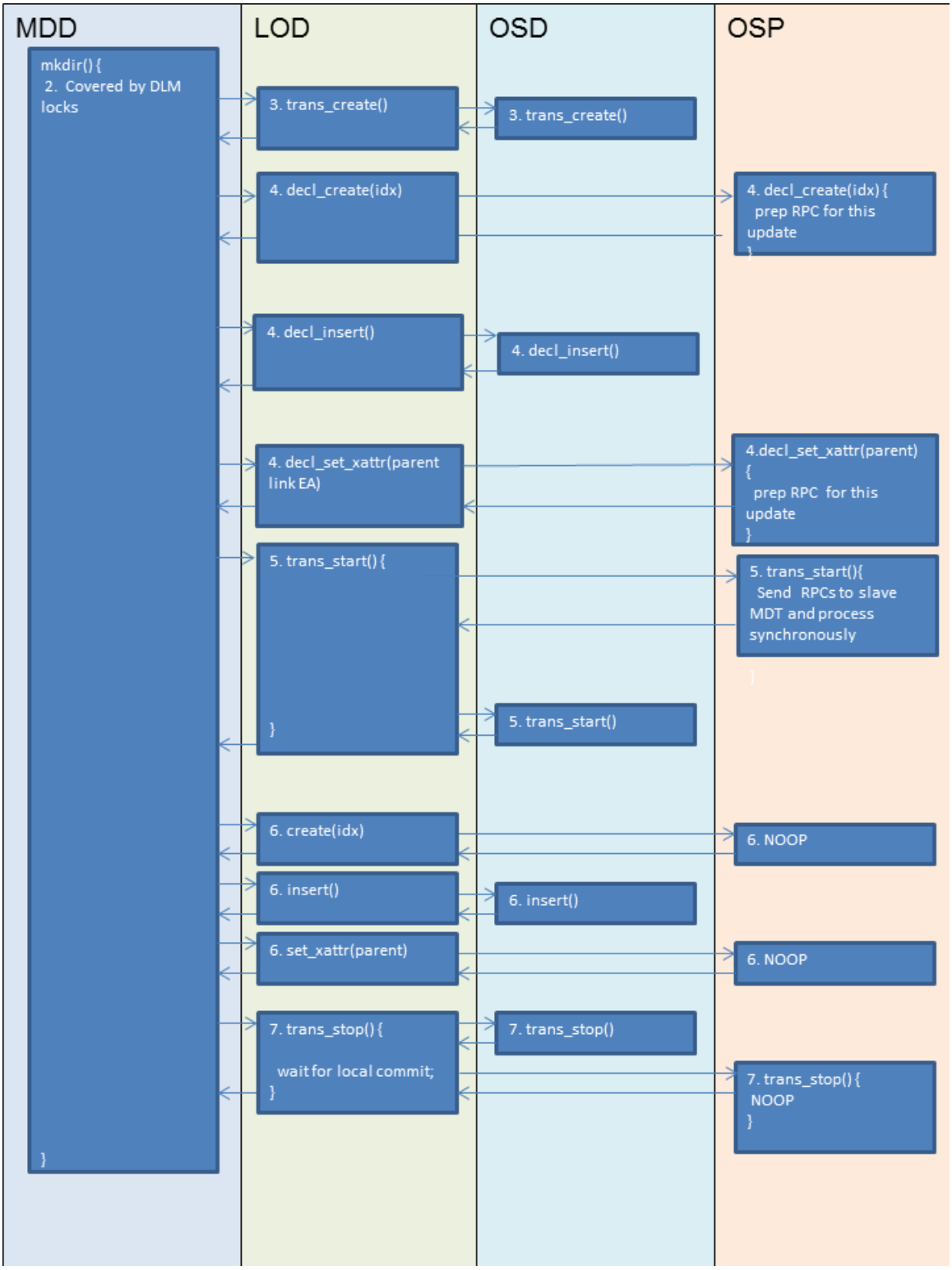
Remote Inode Entry

The local agent inode will reside in the special directories (AGENT DIRs). The agent inode will be located in different agent directories according to which MDT the remote object is located on. For example, if the remote object is located on MDT3, then the local agent inode will be put to AGENT_DIR.3.

Create Remote directory

As described in the Architecture section, any operation on MDT is divided into four transaction steps. For remote directory creation (`lfs mkdir`) the following will occur:

1. The Client allocates a FID for the new directory. The Client will choose the MDT (called the Slave MDT for the operation) given by a parameter to the command. The Client sends the request to the MDT where the parent resides (called Master MDT for the operation).
2. The Master MDT acquires the DLM update lock on the parent directory FID for the name being inserted.
3. On the Master MDT, the MDD layer will create a transaction with the same RPC XID as the Client RPC XID. All the updates for this transaction will be included in the transaction. The MDD decomposes the create operation declares the three object updates:
    a. create the remote directory object, via the Slave OSP, which prepares an RPC for this target.
    b. set_xattr for the remote directory object to store the `link` xattr, via the Slave OSP, which updates the RPC for this target.
    c. insert the name entry into the parent directory, on the Master OSD, which reserves transaction credits this local update, including an update to the `last_rcvd` file to record the Client RPC XID.
4. After all updates have been declared the MDD layer on the Master MDT will start the transaction. The remote create RPC in the OSP layer will be sent to the Slave MDT, where the remote directory object resides, using the FID and attributes supplied by the client. The RPC uses a new XID generated on the Master MDT. This RPC is processed synchronously and is idempotent.
5. The Master MDD executes the local transaction, i.e. insert the name entry into the directory, write the Client RPC XID into the Client's `last_rcvd` slot.
6. In the transaction stop phase, the Master MDD waits for the local updates to be committed to disk, then replies to the Client. This is required to avoid an orphaned remote directory object if both the Master MDT and Client fail simultaneously.



Since all server operations are synchronous there is no need for replay. However the Master MDT may still receive resent mkdir requests from the client and it must handle these as follows:

- If the Master MDT restarts after the name entry is committed (i.e. lost reply to client), the Master MDT tries to match the Client RPC XID against the `last_rcvd` file. If this matches the XID stored in the `last_rcvd` file the entire operation must have completed because the remote directory object was created synchronously on the Slave MDT before the name entry and `last_rcvd` were updated atomically. In this case, the Master MDT can reconstruct the RPC reply from the `last_rcvd` entry.

- If the Client RPC XID is not present on the Master MDT `last_rcvd` file, then the update was not committed on the Master MDT. The Master MDT will resend the same object creation RPC to the Slave MDT using the FID supplied by the Client.
    - The Slave MDT checks if the directory object FID exists. If the object FID does not exist then it is recreated in the same manner as the initial operation.
    - If the Slave MDT does find the object FID then it replies with success to the Master MDT.
    - At this point the Master MDD inserts the name and updates `last_rcvd` as in the normal operation.

Remove remote directory

To remove a directory (`lfs rmdir`), the following will occur:

1. The client sends an unlink RPC operation to the Master MDT, where the parent directory resides.
2. The Master MDT acquires the DLM update lock on the parent directory FID, the lookup lock for the remote directory FID on the Master MDT, and the update and lookup lock for the remote directory FID on the Slave MDT.
3. On the Master MDT, the MDD creates a transaction, then declares the two object updates that make up the unlink operation:
    a. declare destroy of remote directory object via the Slave MDT OSP. This reserves transaction credits for an unlink llog record on the Master OSD, and registers a transaction commit callback to send the destroy RPC to the Slave MDT.
    b. reserving the transaction credits on the Master OSD, to delete the local name entry and update the `last_rcvd` file.
4. When the Master MDT starts the transaction, it will execute the local updates (i.e. delete the name entry and write unlink llog).
5. In the transaction commit phase, the Master MDT waits for the local updates to commit to disk, then the OSP triggers the log sync thread to send the destroy to the Slave MDT. This is the same mechanism as is used for unlinking local files and destroying OST objects upon unlink commit.
6. The destroy on the Slave does not need to be synchronous, because any failure will be handled by llog replay between the Master and Slave MDT.

In case of failure during the remote unlink operation recover proceeds as follows:

- If the Master MDT restarts after the name entry is unlinked and committed (lost client reply), then the resent Client RPC XID will match the entry in the `last_rcvd` file and the reply can be reconstructed.
- If the Master MDT restarts before the name entry is unlinked, the client will resend the unlink RPC. The name entry should still exist on the Master MDT in the parent directory, and the server will redo unlink.
- If the Slave MDT fails before the object destroy is committed, the llog sync thread on the master MDT will resend the destroy record to the Slave. The Slave will destroy the remote directory object if it still exists. This is the same mechanism used to destroy orphan OST objects.

Mode / Attribute Update of remote directory

Changing mode/attributes of the remote directory does not involve its parent, i.e. it is an operation local to the remote MDT directory, so it is the same process as with a single MDT.

1. Client sends an attr_set request using the remote directory FID to the Remote MDT, where the directory inode resides.
2. The Remote MDT acquires the DLM update lock on the remote directory FID, and also the lookup lock if it is changing the permission (uid/gid, acl etc.). Note that usually the lookup lock is used to protect the existence of the name and the permission (uid/gid, acl, etc.) of the inode, and update lock is used to protect other attributes of the inode. For remote directories, due to the split of name and directory, the lookup lock needs to be split as well. The LOOKUP lock on the Remote MDT, where the directory inode is located, will be used to

protect the permission, the LOOKUP lock on the Master MDT where the name entry is located will be used to protect the existence of the name. The DLM update lock will be granted from the Remote MDT.
3. MDD creates a transaction, then declares attr_set updates.
4. MDD executes the updates (changing mode/attributes).
5. MDD commit the transaction and reply to the client. Unlike the remote updates, it does not need to wait for the update to commit to disk, because the Lustre RPC recovery mechanism can handle the failure on a single MDT as usual.

FID on OST

FIDs were designed in a manner that allows existing 1.x filesystems to be upgraded to 2.x while keeping the same inode identifiers. For existing inodes on MDT0 the inode numbers are mapped into the Inode/Generation-in-FID (IGIF) space using the FID sequence range [12-0xffffffff]. Existing OST object IDs are mapped into the ID-in-FID (IDIF) namespace using sequence [0x100000000-0x1ffffffff]. Legacy OST objects are identified by a special surrogate FID called IDIF, which is not unique between OSTs.

In order to maintain compatibility with Lustre 1.x, the current Lustre 2.x implementation limits the filesystem to a maximum of eight MDTs due to the limited space available for the MDT FID numbers that avoid collisions with IGIF and regular FIDs. The FID-on-OST feature needs to be implemented as part of this project in order to allow arbitrary numbers of MDTs. OST FIDs and MDT FIDs will share the same FID space. There is one master FID sequence server in a Lustre filesystem (currently always hosted on MDT0) that will manage the FID allocation and location. The master FID sequence server will sub-assign large ranges of sequence numbers (ranges of 2^30 consecutive sequence numbers called *super-sequences*) to each server in the filesystem, both MDTs and OSTs, which allows parallel and scalable FID allocations from within the super-sequences.

For OST object allocations, the OST is the FID server that requests super sequence FIDs from FID sequence server (MDT0). For OST objects, each MDT behaves as a FID client that requests object FIDs from the OST on which it wants to allocate objects and assigns them to new files as needed. As is currently done for efficiency during MDT file allocation, the MDT pre-allocates multiple OST object FIDs from each OST and caches them locally to reduce the number of RPCs sent to the OSTs.

With the FID on OST feature, the original object ID (l_object_id and l_object_seq in lov_mds_md) will be mapped to a real FID. The OST on which the object resides will still be cached in l_ost_idx in the lov_mds_md, but the OST index could also be derived from only the FID sequence number. To reduce lock contention within each OST, OST objects may be distributed into several directories by the hash of the FID, though this is an internal OST implementation detail. In the current OST implementation, OST objects are stored under the /O/{seq}/[d0-d31]{object_id} namespace (where currently `seq=0` always), and this layout will be maintained with the FID-on-OST feature.

MDT failover and recovery

As with OST failover, MDT failover will ensure that the Lustre filesystem remains available in the face of MDS node failure. By allowing multiple MDTs to be exported from one MDS, Lustre can support active-active failover for metadata as it already does for data. This enables all MDS node resources to be exploited during normal operation to share the metadata load and increase throughput.

Permanent MDT failure

As with data storage, Lustre delegates durability and resilience to its storage devices. Unrecoverable failures in the storage device will result in permanent loss of affected portions of the filesystem namespace. With DNE, this may also make portions of the namespace accessed via the failed MDT inaccessible. The failure of MDT0 is an extreme case which can make the whole filesystem inaccessible.

If all the subsidiary MDTs are referenced directly from MDT0, then the permanent failure of one of these subsidiary MDTs does not affect the others. If there are multiple levels of remote directories then the failure of one MDT will isolate any of its subsidiary directory trees, even though MDT itself is not damaged. Development planned for LFSCK Phase 3 will allow reattaching orphaned directory trees into the MDT0 namespace under a `lost+found-mdtNNNN` directory.

Upgrade Lustre 2.1 to Lustre-DNE

The on-disk format of current Lustre 2.x filesystems will be forward compatible with DNE on MDT0 and on the OSTs, so existing Lustre filesystems can be upgraded to a DNE-capable version of Lustre without affecting the data therein. Just upgrading to a DNE-capable version of Lustre will not affect the on-disk filesystem format. When a DNE-capable version of Lustre is running on both the clients and servers, new MDTs can be added to the existing filesystem while it is mounted and in use. The newly added MDTs will not be utilized until a remote directory is created there. Once a remote directory is created on a subsidiary MDT using `lfs mkdir`, it will not possible to directly downgrade to a version of Lustre that does not support DNE. At this point, clients that do not support DNE will be evicted from the filesystem and will no longer be able to access it.

If the user wants to downgrade Lustre on the MDS from a multi-MDT filesystem to an older version of Lustre that does not support DNE (i.e. a single-MDT filesystem), all files and directories must be copied to MDT0 before removing the other MDTs. The configuration must be rewritten without allowing the subsidiary MDTs to reconnect to the MGS. Without these steps, any directories and files not located on MDT0 will be inaccessible from the old Lustre MDS.

Use Cases

Upgrade single MDT to DNE.

1. All Lustre servers and clients are either 2.1 or 2.2. Lustre on-disk format must be 2.1 or 2.2.
2. Shutdown MDT and all OSTs, then upgrade MDT and all OSTs to Lustre version with DNE. Remount MDT and OSTs.
3. Adding new MDT by

```
mkfs.lustre  --reformat –mgsnode=xxx –mdt --index=1 /dev/{mdtn_devn}
 mount -t lustre -o xxxx /dev/mdtn_devn /mnt/mdtn
```

4. Client who does not understand DNE will be evicted.
5. Upgrade clients to Lustre version with DNE.

Downgrade DNE to single MDT

1. Construct a Lustre filesystem with four MDTs.
2. The administrator move all of directories and files from MDT1, MDT2, MDT3 back to MDT0, and disable MDT1, MDT2, MDT3.
3. Shutdown MDTs and all of OSTs. then downgrade DNE to prior DNE version.
4. Remount MDT and OSTs and start with single MDT.
5. Clients that do not understand DNE (1.8, 2.1, 2.2) will be evicted.

Implementation Milestones

The Implementation phase is divided into the following three milestones, each with an estimated completion date:

Implementation Milestone 1

Demonstrate working DNE code. The sanity.sh and mdsrate-create tests will pass in a DNE environment. Suitable new regression tests for the remote directory functionality will be added and passed, including functional Use Cases for upgrade and downgrade. (April 2th 2012)

Implementation Milestone 2

Demonstrate DNE recovery and failover. Suitable DNE-specific recovery and failover tests will be added and passed. (July 30th 2012)

Implementation Milestone 3

Performance and scaling testing will be run on available testing resources. The Lustre Manual will be updated to include DNE Documentation. (Sep 7th 2012)