LFSCK 3 MDT - MDT Consistency High Level Design

1 Introduction

Lustre has supported Distributed NamespacE (DNE) mode since the Lustre-2.4 release. Compared with the original single-MDT configuration, there is additional infrastructure and new operations used with DNE that may introduce new inconsistencies between multiple MDTs that need to be repaired by LFSCK. The main DNE components that are introducing new functionality are:

- **LMV**: The Logical Metadata Volume (LMV) is the client-side subsystem that aggregates multiple MDTs into a single virtual metadata server for the client. It determines at the client which MDT holds the metadata for a particular object by the object's FID and the FID Location Database (FLD) mapping table. DNE uses the `XATTR_NAME_LMV` extended attribute (xattr) for a number of purposes including storing layout information for striped directory and storing some flags for metadata migration. Each kind of LMV xattr has a unique 'magic' number. The type of an LMV xattr can be determined from the value of its 'magic' member. See the definition of the union type lmv_mds_md for details. The following two 'magic' values are used to distinguish between the master LMV xattr and the slave LMV xattr:
    - **`LMV_MAGIC`:** This LMV xattr stores the layout for a striped directory on the master MDT-object. It describes the number of directory stripes/slaves that make up the directory, and the hash function that distributes filenames across the stripes.
    - **`LMV_MAGIC_STRIPE`**: The LMV xattr that stores the slave MDT-object information on the slave MDT-object of striped directory, such as the hash function, the stripe index, and the total number of stripes.
- **Distributed name entry and MDT-object**: With multiple MDTs, a given directory on one MDT can contain a name entry that references an MDT-object on a remote MDT.
    - From the name entry point of view, the MDT-object is a **remote MDT-object**. On the local MDT, the name entry references the remote MDT-object's via the triple <name, FID, type>.
    - From the MDT-object view, the name entry is a **remote name entry**. On the local MDT, the MDT-object points back to the remote name entry via its linkEA xattr entry (parent directory FID + the child name).
- **Striped directory**: Similar to a Lustre striped file, a directory also can be striped across multiple MDTs under DNE mode. A striped directory contains one master MDT-object and two or more slave MDT-objects. Each slave MDT-object maintains its own nlink count attribute, and the whole directory's nlink count is the sum of all slave MDT-objects' nlink count.
    - The master MDT-object stores the stripe information as sub-directories in the master directory**.** Each name entry (sub-directory) references one slave MDT-object. Other attributes, such as hash types are stored in the master LMV xattr (with **`LMV_MAGIC`**).
    - The slave MDT-object references the master MDT-object via the ".." entry. To indicate that it is a slave MDT-object, it also stores a slave LMV xattr (with **`LMV_MAGIC_STRIPE`**).
    - For the client, when a new file is created under a striped directory, it will apply the hash function for that directory to the filename (modulo the number of directory stripes) to select the slave MDT-object into which the name entry will be inserted. Clients do not see the slave directories in their namespace, only the master directory.

In LFSCK 3, we will extend both the FID-in-dirent and linkEA verification from single-MDT to cross-MDT cases, and also add further check and repair for the following DNE inconsistencies:

- **Dangling name entry**: The name entry exists, but related MDT-object does not exist.
- **Orphan MDT-object**: The MDT-object exists, but there is no name entry to reference it.
- **Multiple-referenced name entry**: More than one MDT-object points back to the same name entry, but the name entry only references one of them.
- **Unmatched name entry and MDT-object pairs**: The name entry references an MDT-object which has no linkEA for back-reference, or it points back to another name entry that does not exist or does not reference the MDT-object.
- **Unmatched object type:** The file type stored in the directory entry does not match the file type stored in the MDT-object attribute.
- **Invalid nlink count**: The MDT-object's nlink count does not match the number of name entries which reference such MDT-object.
- **Invalid name hash for striped directory**: A name entry on a slave MDT-object (shard) hashes to an index that does not match the index stored in the slave object's LMV xattr.

LFSCK 3 needs to find and repair the above inconsistent cases while the system is running with normal client usage of the filesystem. The design and implementation will be against the master branch and should be OSD neutral.

2 Discover MDT-MDT inconsistency

The LFSCK for MDT-MDT consistency needs to scan the whole system on all MDTs to discover inconsistencies. Completeness and efficiency are the main requirements for LFSCK system scanning. To achieve comprehensive scanning of the whole system, we will enhance the existing two-stage scanning method, and for each stage, the system scanning will run on all MDTs in parallel.

2.1 First-stage scanning - verify name entries

The first-stage system scanning will be driven by the lower-layer object-table based iteration and incremental namespace based directory traversal. This is done via the same scanning pass as described in the LFSCK 1.5 and LFSCK 2 High Level Designs to minimize LFSCK overhead.

The work for lower-layer object-table based iteration is:

- If the target MDT-object is a regular file with multiple links (nlink count greater than 1, or the linkEA xattr contains multiple entries), or if it has a remote name entry (determined by checking the parent FID, hereafter referred to as PFID, in the linkEA), then record it in the LFSCK tracing file (see section 3) for further processing.

The work for namespace based directory traversal is:

- If the target MDT-object is a directory, then check whether it is striped directory or not via the presence of the LMV xattr. If yes:
  - verify the shard entries in the master directory
  - verify that the name of each entry is correctly mapped to its shard using the LMV hash function
- Check whether the MDT-object exists or not. If yes, check whether it has a valid linkEA back reference or not.
- For non-striped directories, or shards of striped directories, verify the name entry and child FID, and the linkEA on each child object as with LFSCK 1.5.

When the MDT first-stage system scanning has completed, the following MDT-MDT inconsistencies are identified:

- Lost LMV xattr
- Invalid name hash for striped directory
- Dangling name entry
- Unmatched object type

2.2 Second-stage scanning - verify multiple-linked files and orphan MDT-objects

The second-stage scanning is almost the same as the LFSCK 1.5 second-stage scanning which handles multiple-linked files. This second-stage scan will be done after the first-stage scan finishes on all MDTs. The first-stage of LFSCK scanning will have recorded all the multiple-linked files and the files with remote name entries in the LFSCK trace file. The LFSCK can use the existing `dt_index_operations` APIs to scan all multiple-linked/remote MDT-objects, and for each one, verify its linkEA entries one by one:

- Invalid or redundant name entries will be removed.
- The MDT-object's nlink count will be verified.
- Remote orphan MDT-objects will be found and repaired.
- Unmatched name entry and MDT-object pairs
- Multiple-referenced name entries will be found and repaired.

Another kind of orphan MDT-object is special for an ldiskfs-based backend. The local e2fsck will insert local orphans into a local `/lost+found` directory. Local e2fsck does not understand the linkEA, so it cannot repair those disconnected inodes beyond inserting them into `/lost+found`, but LFSCK can reconnect the inode into the proper parent directory if it is available. LFSCK will process entries in `/lost+found`, and for the objects that have valid `link` xattrs, LFSCK will try to link them back to their original place in the namespace. According to the parent directory's FID in the linkEA, LFSCK knows in which MDT the name entry should reside.

3 LFSCK tracing

The LFSCK 3 will share the existing on-disk file **`lfsck_bookmark`** to store some parameters, and will extend the existing on-disk lfsck trace file **`lfsck_namespace`** to trace both the LFSCK 1.5 and LFSCK 3 processing. See the LFSCK 1.5 High Level Design for the description of the **`lfsck_namespace`** trace file. The status, statistics, checkpoint, and so on, will be recorded in this file. It can be used to query lfsck processing status from user space, and also for resuming the LFSCK 3 from breakpoint.

- **`lfsck_namespace::ln_magic`**
  Extending the existing `lfsck_namespace` file will cause compatibility issues when the MDT downgrades to old Lustre-2.x (x <= 6). To avoid confusing the old LFSCK 1.5, the LFSCK 3 will use a new magic number (**`LFSCK_NAMESPACE_MAGIC_V2`** for the extended **`lfsck_namespace`** file). Then after a downgrade, the old LFSCK 1.5 will not recognize the new magic number, so it will reset the **`lfsck_namespace`** file. Similarly, when the file system is updated from old LFSCK 1.5 to LFSCK 3, the **`lfsck_namespace`** file will be reset. Since the **`lfsck_namespace`** file only stores restart information for a single LFSCK scan, at worst this will cause a new full scan the next time LFSCK is run.

3.1 Record failed servers during the first-stage scanning

To check MDT-MDT consistency verification during the first-stage scanning, if LFSCK on the MDT wants to verify some name entry for a remote MDT-object but related MDT is unavailable at that time (maybe because of a bad network or if the MDT is restarted), the failed MDT will not know whether the related MDT-object is referenced by some name entry or not. If the MDT-object stores an invalid linkEA, then it will misguide LFSCK to take some incorrect behavior when it handles orphan MDT-objects. Consider the scenario that LFSCK on the MDT0 is scanning the directory_A, and finds that the name entry_B references a remote object_C which resides on MDT1, then LFSCK on the MDT0 will trigger RPC to the MDT1 to check whether object_C exists and whether the object_C back reference the name entry_B via its linkEA. But at that time, such verification failed because that the MDT1 is unavailable temporarily (or MDT1 is rebuilding OI files and the FID mapping for object_C is not ready yet). Then when LFSCK moves to the second-stage scanning, the MDT1 does not know it has ever missed/failed to verify the remote name entry_B for the object_C, it will handle the object_C as following:

1. If the object_C has no linkEA, then it will be regarded as orphan by the LFSCK on the MDT1 during the second-stage scanning.
2. If the object_C has an invalid linkEA with parent object_D and name_E, then the LFSCK on the MDT1 will do:
    a. If the object_D does not exist, then the object_C will be regarded as orphan. Otherwise,
    b. If the name_E does not exist under the object_D, then the name entry_E will be added. Otherwise,
    c. it will be handled as multiple-reference name entry case.

All above cases are improper repairing behavior. To avoid that, LFSCK on each MDT will maintain a bitmap in its own LFSCK tracing file (**`lfsck_namespace`**) to record which MDTs have ever missed/failed to verify some name entries during the first-stage scanning. Then LFSCK on this MDT can notify the other ever failed MDTs to skip orphan MDT-object handling in the second-stage scanning. The possible skipped orphan MDT-objects will be handled the next time LFSCK is run. We do not think that it is normal that the same MDT will always become unavailable during the first-stage scanning for every LFSCK run.

4 LFSCK user space control

We will try to reuse the existing LFSCK user space tools to control LFSCK for MDT-MDT check/repair by specifying the "`-t namespace -A`" options. These options tell Lustre to start the namespace LFSCK on all MDT devices.

```
lctl lfsck_start –M lustre-MDT0000 –t namespace -A
```

There is no need for a new LFSCK component type "`dne`" as was done for previous LFSCK phases. DNE mode is the namespace extension from single MDT to multiple MDTs, so there would be no clear semantic difference between "`-t dne`" and "`-t namespace -A`". To avoid confusion, the DNE remote and striped directory checking will be done as part of the LFSCK 1.5 namespace checking. The LFSCK 3 will also share (and maybe enhance) other interfaces with LFSCK 1.5.

The administrator can use the following interface to query LFSCK processing:

```
 lctl get_param mdd.$fsname-MDT${idx}.lfsck_namespace
```

The administrator can specify the max speed for the LFSCK to scan the device through the following interface:

```
lctl set_param mdd.${fsname}-MDT${idx}.lfsck_speed_limit=N
```

5 LFSCK engines

The MDT-MDT consistency check/repair is driven by a series of kernel threads, including a master engine and an assistant thread on every MDT.

5.1 LFSCK master engine

There is a local LFSCK master engine on every MDT, it works as follows:

1. When the master engine on the MDT is triggered by the LFSCK user space command (`lctl lfsck_start`), send LFSCK control RPCs to trigger the master engines on all other MDTs.
2. Start the LFSCK assistant thread locally.
3. Start the first-stage system scanning on the MDT through lower-layer otable-based iteration + namespace-based directory traversal. Give the name entries to the local LFSCK assistant thread for verification as described in the next section.
4. Notify the LFSCK assistant thread that the first-stage scanning on the MDT is done, no more inputs for the pipeline.
5. Wait for the LFSCK assistant thread to finish the first-stage verification, then trigger the second-stage scanning locally.

5.2 LFSCK assistant thread

For performance reasons, LFSCK 3 uses an assistant thread on every MDT to verify the name entries together with the master engine, which feeds name entries into a pipeline. The master engine fetches the name entries from the directory, then pushes them into the pipeline. The LFSCK assistant thread verifies the consistency of each name entry one by one from the pipeline. This allows the master engine to drive all the LFSCK components efficiently, without being blocked by network issues or a bad peer. The LFSCK assistant thread works as follows:

1. Start when triggered by the master engine locally.
2. For each name entry in the pipeline, verify its consistency.

3. Send an LFSCK control RPC to all other MDTs to notify them that the first-stage scanning on the MDT is finished.
4. Wait for all other MDTs to finish the first-stage system scanning.
5. Wait for the master engine to announce the second-stage scanning, then scan orphan MDT-objects and multiple-linked files. Since there will not be too many distributed name entries and MDT-object pairs, do the second-stage scanning synchronously.
6. Notify the master engine and exit.

6 Repair MDT-MDT inconsistency

NOTE: this section only describes the check/repair logic. API and RPC changes are discussed in subsequent sections.

6.1 Repair dangling name entry

With the given FID (in dirent or in LMA), LFSCK can know on which MDT the target MDT-object resides. If the target MDT-object does not exist on such MDT, then handle it as the administrator wants, two options:

1. Keep the name entry there and report inconsistency. (By default, the same policy as LFSCK uses for repairing the MDT-object with dangling reference in the LOV EA.)
2. Recreate the lost MDT-object, and store the name in the linkEA. To ensure the re-created MDT-object is distinguishable from normal create cases, the MDT-object's ctime will be set to zero. If someone modifies the MDT-object after it was re-created, the ctime will be updated. If the lost MDT-object is a shard of striped directory, then also generate the slave LMV EA for it.

6.2 Repair unmatched name entry and MDT-object pairs

1. If the MDT-object is not a directory, then:
    a. If the MDT-object has no linkEA, then generate the linkEA. Otherwise,
    b. If the MDT-object's linkEA does not match the name entry, then during the first-stage scanning, the LFSCK cannot know whether the linkEA is valid or not, so the LFSCK will add another entry in the linkEA according to the name entry. Then during second-stage scanning, according to the rules in section 6.5.1, the linkEA entries will be verified, and the unrecognized linkEA entries will be removed.
2. If the MDT-object is a directory, then the LFSCK will check the MDT-object ".." entry.
    a. If the ".." entry references the parent directory of the name entry, then the name entry is trusted:
        i. If the target MDT-object has no linkEA, then generate linkEA according to the name entry. Otherwise,
        ii. If the linkEA entry back references another parent directory, then the linkEA entry is invalid and will be removed, the LFSCK will generate new linkEA entry according to the name entry. Otherwise,
        iii. If the linkEA entry is recognized, then the name entry is invalid and will be removed, the parent's nlink count will be decreased.
    b. If the ".." entry does not reference the parent directory of the name entry, then during the first-stage scanning, the LFSCK may cannot know whether the linkEA is valid or not, so the LFSCK will add another entry in the linkEA according to the name entry. Then during second-stage scanning, according to the rules in section 6.5.2, the linkEA entries will be verified, and the unrecognized linkEA entries will be removed.

NOTE: If LFSCK removes an invalid name entry that is for a shard of a striped directory, then there will be a hole in the master LMV EA. That is acceptable, just like the LOV EA hole for repairing an orphan OST-object. We expect that, as LFSCK progresses, the original real shard of the striped directory will be found, at which point LFSCK can re-insert the right name entry to the master object according to the slave object (for a striped directory, the name for a shard is composed of `$FID:$mdt_index`). However, before that point and before the LFSCK re-creates the MDT-object for dangling name entry (according to the LFSCK start options), accessing the striped directory with an LMV EA hole may return an error.

6.3 Repair multiple-referenced name entry

This inconsistency is probably caused by an interrupted rename operation or migration. The LFSCK will check whether the conflicting MDT-object (that is recognized by the name entry) was created by LFSCK attempting to repair a dangling name entry by checking the MDT-object ctime attribute. If the MDT-object has a ctime of zero, this means that nobody has updated the MDT-object after its creation, so LFSCK will destroy the conflicting MDT-object and update (delete + insert) the name entry to reference the current MDT-object. Otherwise, the conflict MDT-object may contain valid data, and we cannot remove it. In this case, it is not clear whether the un-recognized MDT-object is useful or not, so it is better NOT to destroy it:

1. If the linkEA entry is not the unique one for the un-recognized MDT-object, the remove the linkEA entry; otherwise,
2. The LFSCK will generate a new local name entry under the directory `.lustre/lost+found/MDTxxxx/` with the name `$PFID-O-$conflict_version`. The LFSCK will update the unrecognized MDT-object's linkEA to match the new parent and name entry.

NOTE: For the shard of a striped directory, its name is composed of `$self_FID:$mdt_index`. Thus, for a given name entry for a shard object of the striped directory, the LFSCK can directly determine whether the name entry is corrupted or not. Even if the name entry contains corrupted data, it is almost impossible that the name and FID information stored in the name entry are corrupted in the same way. If the name entry is valid, and if it is dangling, then the re-created MDT-object for the lost shard object will NOT be wrong repairing. Thus we can assume that the unrecognized MDT-object for the multiple-referenced name entry cannot be the original lost shard object. Furthermore, if the unrecognized

MDT-object is a shard object (i.e., it has a slave LMV xattr) of some striped directory, it must belong to another striped directory, so the LFSCK will create a new master object to reference the orphan shard.

6.4 Repair orphan MDT-object

During the second-stage scanning, for every object in the LFSCK tracing file, the LFSCK will do the following:

1. If the object has no linkEA, then generate a new name entry under the directory `.lustre/lost+found/MDTxxxx/` with the name `$FID-O-$conflict_version` locally.
2. If the object has a remote linkEA entry, then check whether the remote name entry exists in the parent directory or not:
    a. If the parent directory does not exist, then the LFSCK will send an RPC to the remote MDT to re-create the parent under `.lustre/lost+found/MDTxxxx/` with the name `$FID-P-$conflict_version`, and then insert the name entry according to the linkEA entry into the new re-recreated parent remotely; otherwise,
    b. If the target name entry does not exist, then the LFSCK will insert the name entry and increase the parent directory's nlink count (if it is a directory object) remotely; otherwise,
    c. If the remote name references another MDT-object, then it falls under the multiple-referenced name entry case and will be repaired as described above.

By recreating a missing parent FID in `.lustre/lost+found/MDTxxxx/` it is possible for the LFSCK to entirely reconstruct a lost directory, including the names of all the entries, based on the linkEA entries in each of the child files or subdirectories. If that directory's parent entry also exists, the LFSCK will find it and reconnect the entry from `MDTxxxx/` into the namespace when the LFSCK run next time.

If the orphan is a shard of a striped directory, then it is possible that the master object of the striped directory is lost. Then the LFSCK will re-create the master object as described above and re-generate the master LMV EA according to the slave object. At that time, because the LFSCK does not know about the other shard(s), there may be holes in the LMV EA. That is acceptable, just like the LOV EA hole for repairing orphan OST-object. As the LFSCK continues, more orphan shards may be found to fill the empty LMV EA slots.

6.5 Verify multiple referenced MDT-object - invalid/redundant linkEA entries, invalid nlink count

For a POSIX filesystem, only regular files can have multiple links. Under some inconsistent cases, it is possible that multiple name entries may reference the same directory. The LFSCK needs to handle multiple references for both file and directory MDT-objects.

6.5.1 multiple-referenced normal file

During the second-stage scanning, for each MDT-object recorded in the LFSCK local trace file, the LFSCK will check all its linkEA entries: 'M' is the count of linkEA entries, 'N' is MDT-object's nlink count. Here, 'M' and/or 'N' is greater than 1, and 'M' is not smaller than the count of the known name entries after the successful first-stage scanning. For each linkEA entry, the LFSCK will do the following:

1. If the linkEA entry contains invalid information, such as an invalid FID, then remove it directly, and keep the MDT-object's nlink count unchanged.
2. If the linkEA entry is repeated (redundant), then destroy the linkEA entry, and keep the MDT-object's nlink count unchanged.
3. If the name entry corresponding to the linkEA entry does not exist, and:
    a. If 'M' > 'N', then remove the linkEA entry, and keep the MDT-object's nlink count unchanged.
    b. If 'M' <= 'N', then add the new name entry according to the linkEA entry, but keep the MDT-object's nlink count unchanged.
4. If the name entry corresponding to the linkEA entry exists, but it references another MDT-object:
    a. If the conflicting MDT-object was created by the LFSCK for repairing a dangling name entry (ctime attribute is zero), then destroy the conflict MDT-object and update (delete + insert) the name entry according to the linkEA entry, and keep the MDT-object's nlink count unchanged. Otherwise,
    b. Remove the linkEA entry, and if it is the last linkEA entry, then add a name entry for the orphan MDT-object to .lustre/lost+found/MDTxxxx/, and set the MDT-object's nlink count to 1.
5. After all the linkEA entries have been verified, then verify that the nlink count, 'N', is equal to the number of linkEA entries, 'M'. If 'M' is not equal to 'N', then trust 'M', and set the MDT-object's nlink count to 'M'.

6.5.2 multiple-referenced directory

Usually, for a local file system, a multiple-referenced directory object can be repaired by some local filesystem consistency verification tools, such as e2fsck for the ldiskfs backend. But under DNE mode, the name entry for the directory may be remote. The LFSCK will verify the directory MDT-object's ".." entry firstly:

1. If the parent directory has some name entry to reference the target MDT-object, then the "parent FID + the name entry" is the unique valid linkEA entry, and all other linkEA entries and related name entries will be removed. Otherwise,
2. If the parent directory has no name entry to reference the target MDT-object, then the LFSCK will check every linkEA entry in turn. If some parent directory recognize the target MDT-object, then such name entry will be used for the unique valid linkEA entry, and all other linkEA entries and related name entries will be removed. The target MDT-object's ".." will be updated also.
3. If all linkEA entries have been verified as invalid, then it will be regarded as orphan, and moved to the `.lustre/lost+found/MDTxxxx`

`/` with the name `$FID-O-$conflict_version`.

6.6 Repair invalid file type

Generally, the LFSCK will trust the file type claimed by the MDT-object (stored in its attribute), so the name entry should be fixed, whether it is a local name entry or a remote one. Because there is no dt_object "update" API, the LFSCK will combine "delete" and "insert" to update the type in the name entry.

6.7 Repair invalid name hash for striped directory

The name hash verification depends on the valid LMV EA. So the LMV EA will be verified firstly. For the LFSCK, both the striped directory and each of its shards are directory objects. The LFSCK will basically scan them as it scans normal directories. On the other hand, the LMV EA provides additional clues that can be used to locate and repair the inconsistency. During otable-based iteration in the first-stage scanning, the LFSCK engine will check whether the MDT-object is a striped directory, a shard of a striped directory, or neither. These are the general rules:

R1: The name for a sub-directory entry in a striped directory (master) object is of the form `$FID:$mdt_index`. Names of any other form are invalid.

R2: Only slave objects (shards) are allowed under a striped directory. All others, such as normal files, normal directories, and so on, are invalid.

6.7.1 The master LMV EA matches the slave LMV EA

This is the normal case. Both the master LMV EA and the slave LMV EA can be corrupted, but it is almost impossible that the two copies have the same data corruption. So we trust the LMV EA. On the slave object side, for each name entry, the LFSCK will recalculate its name hash. If the hash does not match the MDT, then there are two possible cases:

1. The name entry is corrupted. Check whether the MDT-object referenced by the name entry (via the FID) exists and whether it back references the name entry via its linkEA. If not, then the name entry will be regarded as corrupted, and it will be removed. There is no need to worry about losing data, because if the target object (single linked) still exists, then the real name can be recovered when the LFSCK handles orphan MDT-objects. Otherwise,
2. Since some MDT-object still back-references the name entry, it is possible that the name entry was inserted on the wrong MDT. The LFSCK will move the name entry to the right MDT.

6.7.2 The directory object has no LMV EA

If the directory object has no LMV EA, then it is quite possible that it is a normal non-striped directory. But it is also possible that the striped directory or its shard lost the LMV EA. At that time, the LFSCK will verify the MDT-object as normal directory. If it is the case that the MDT-object is a striped directory, then since there are 1+N copies for the LMV EA, as long as 1 copy is still intact, then as the LFSCK continues, some other LFSCK instance running on another MDT will discover that the master (or slave) MDT-object lost the master (or slave) LMV EA. Then it will re-generate the lost LMV EA and notify the LFSCK instance on this MDT to re-scan the directory for LMV-EA-related verification. For an LFSCK instance running on `MDT_X`, it is unpredictable when another LFSCK instance on `MDT_Y` might send the RPC to re-generate the lost LMV EA and notify `MDT_X` to re-scan the striped directory (or the shard of a striped directory). Thus, LFSCK will maintain a **rescan_list** on each MDT for DNE LFSCK:

1. When LFSCK receives notification to rescan some MDT-object, the MDT-object will be linked into the **rescan_list.**
2. Anytime when LFSCK on the MDT has finished the LMV EA and name hash verification for the MDT-object, the MDT-object will be unlinked from the **rescan_list.**
3. When the first-stage scanning has completed, LFSCK on the MDT will handle all the MDT-objects on the local **rescan_list** one by one for LMV EA related verification as described below.

6.7.3 The master LMV EA and the slave LMV EA does not match

1. If LFSCK on the slave finds that the slave LMV EA does not match the master LMV, then it cannot know which one is correct. It will skip the name hash verification (as described in the section 6.7.1). When the LFSCK on the master discovers the right LMV EA (see 6.7.3.2), it will notify this LFSCK instance to rescan this shard for name hash verification.
2. If LFSCK on the master finds that the master LMV EA does not match the slave LMV EA, then it will try to check the next slave LMV EA. The master object will scan all the slaves' LMV EA:
    a. If two of them (master LMV EA or slave LMV EA) match each other, then the LFSCK can trust them, and it will repair other non-matched LMV EA, and notify the related LFSCK instance to rescan related shards for name hash verification.
    b. If none of the slaves' LMV EA and master LMV EA match one another, then the LFSCK cannot know which LMV EA hash type is correct. Under such bad case, then LFSCK will change the LMV EA hash type to `LMV_HASH_TYPE_UNKNOWN`. Then the client can treat striped directories with that special hash type as read-only. In this case, the client can access all of the files in the striped directory. These files should be renamed to another striped directory to be rehashed. This will require that it be possible to unlink files from a directory with unknown hash type.

7 Object visibility changes

The LFSCK needs to access some OSD internal objects for the orphan repairing.

7.1 Export `/lost+found`

The `/lost+found` directory is an ldiskfs local directory to hold the local unreferenced inodes. LFSCK needs to scan this directory to move the local orphans back to client visible namespace, so we assign a special FID to it to make it to be visible to LFSCK: `{ FID_SEQ_LOCAL_FILE, LU_BACKEND_LPF_FID, 0 }`. For ZFS-based backends, there is no local `/lost+found` directory. LFSCK will get an `-ENOENT` error when trying to access the `/lost+found` directory. There will be no compatibility issues with old MDT devices.

8 API changes

Enhance some existing APIs for the DNE LFSCK.

8.1 Enhance `dt_index_operations::dio_insert`

The insert() method is mainly used for adding the name entry to the parent directory. Usually, the parameter "@rec" is the target MDT-object's FID (appended after the name entry or in LMA), and the parameter "@key" is the name, but the target MDT-object's type (normal file or directory) is not passed from the caller. According to the current DNE implementation, the OSD assumes that the remote MDT-object is a directory, so the target type will be set as "directory" by default. Such an assumption for a striped normal file (the name entry and MDT-object reside on different MDTs) is wrong. It may not cause visible failure for normal DNE operations, but for LFSCK, such an assumption is not acceptable. LFSCK needs to know the target type correctly and use it to verify (and/or re-create) the remote MDT-object.

```
int (*dio_insert)(const struct lu_env *env, struct dt_object *dt,
                          const struct dt_rec *rec, const struct dt_key *key,
                          struct thandle *handle, struct lustre_capa *capa,
                          int ignore_quota);
```

So the "@rec" parameter will be the pointer to the following structure:

```
struct dt_insert_rec {
 union {
  const struct lu_fid *fid;
  ...
 };
 union {
  __u32    type;
  ...
 };
};
```

8.2 New `LFSCK_NOTIFY` event - `LE_CREATE_ORPHAN`

If the LFSCK wants to create an orphan object remotely (for example to create the lost parent for some linkEA entry), then it will do that via the `LFSCK_NOTIFY` RPC with the new LFSCK_NOTIFY event: LE_CREATE_ORPHAN. We do not use the name create method, because from the local LFSCK view, it does not know the parent object (.lustre/lost+found/MDTxxxx/) to hold the orphan. Furthermore, such parent object may not have been created yet. The normal cross-MDT create operation cannot handle these cases well.

- **`LE_RESCAN_DIR`**

```
enum lfsck_events {
        LE_LASTID_REBUILDING    = 1,
...
  LE_CREATE_ORPHAN  = 12,
};
```

8.3 New **`LFSCK_NOTIFY`** event - **`LE_RESCAN_DIR`**

As described above, when LFSCK re-generates the master/slave LMV xattr and sets it remotely via `OUT_XATTR_SET` RPC, it will also notify the LFSCK instance on the remote MDT to rescan the target MDT-object to verify the stripe's slave LMV xattr or children name hash. Such a notification will be triggered by the `LFSCK_NOTIFY` RPC with the new LFSCK_NOTIFY event: `LE_RESCAN_DIR`.

- **`LE_RESCAN_DIR`**

```
enum lfsck_events {
        LE_LASTID_REBUILDING    = 1,
...
  LE_RESCAN_DIR   = 13,
};
```

9 Race control between MDT-MDT consistency check/repair and other operations

Generally, the DNE LFSCK will be lockless during a routine check to avoid lock conflict with other normal namespace operations. In the most common cases, the objects being checked are static, and there will not be any inconsistencies found. In the rare case that LFSCK finds an inconsistency, it needs to take the same LDLM locks on the MDT as normal DNE operations on the affected object(s) and re-check the target to distinguish the inconsistency from normal modifications to the object. For example, a cross-MDT create will create an MDT-object on `MDT_X` first, and then insert the name entry into the parent directory on `MDT_Y`. If LFSCK finds the newly created MDT-object between these two steps it may regard the MDT-object as an orphan and try to repair it. Also, consider the async mode DNE operations which are to be supported: the name entry may be inserted before the MDT-object is created, and then it may be regarded as dangling name entry by the DNE LFSCK. Holding the LDLM locks on the affected objects is necessary to exclude normal filesystem operations before LFSCK re-verifies and repairs any inconsistency. To avoid deadlock among the LFSCK and other normal DNE operations, and also among LFSCK instances on different MDTs, the lock order must follow the same lock order as other normal cross-MDT operations. (The general rules: lock parent before the child, and lock source before the target.)

9.1 Handle orphan MDT-object and cross-MDT create

During the first-stage scanning, if LFSCK finds a directory object, it will check the parent with its ".." name entry. If the object has no ".." entry yet, then it is quite possible that it is a half-created MDT-object for cross-MDT operation as described above. In this case, LFSCK will take LDLM lock on the object and recheck. If it is really for cross-MDT operation, then when the LFSCK moves to second-stage scanning, the ".." entry should have been added before the LDLM locks held by the creating thread are dropped, so it is harmless. Otherwise, it is a corrupted directory object and LFSCK will move it to `.lustre/lost+found/MDTxxxx/`.

9.2 Orphan MDT-object verification and metadata migration

When performing metadata migration, there will be a new `MDT-object_A` created on `MDT_X` firstly. Then the related name entry under the parent `MDT_object_B` on the `MDT_Y` will be updated to reference the new `MDT-object_A`, and then the old `MDT-object_C` will be destroyed. All these sub-operations are not in single transactions, and there will be some interval that the LFSCK finds the newly created `MDT-object_A` during the migration. To prevent the LFSCK from regarding the newly created `MDT-object_A` as an orphan, the migration thread should set some flags on the newly created `MDT-object_A` within the same transaction when creating the `MDT-object_A`. The linkEA for the `MDT-object_A` is the first and normal choice for those flags. With the help of the LDLM lock on the old `MDT-object_C`, the LFSCK can know that the `MDT-object_A` is the migration target.

9.3 Name entry verification and rename

If rename happened before the name entry is verified, then the rename operation will guarantee that the new name entry is inserted into the right parent directory (name hash) and the linkEA will be updated also. If the target MDT-object is a multiple-linked file, other aliases will be verified via other name entries by the LFSCK normally.

During the first-stage scanning, if the object has a ".." name entry, then verify whether the ".." references the current directory or not.

1. For normal case, the ".." will be the current directory. Otherwise,
2. Record the object in the LFSCK tracing file for further verification during the second-stage scanning.

10 Recovery process

It is inevitable that one or more MDTs may experience an error condition during DNE LFSCK on large file systems. To be a robust file-system consistency verification tool, the DNE LFSCK must handle such cases properly.

10.1 Recovery from the first-stage scanning failures

If the MDT crashes during the first-stage scanning, before it's restart, other MDTs may need to communicate with the failed MDT for remote MDT-objects verification. Because it cannot respond, those LFSCK instances cannot verify related name entries. So the other MDTs will record the failed MDT in their local LFSCK trace files "`lfsck_namespace`" and go ahead to handle other objects without being blocked by the failed MDT. It also may be fortunate enough that no other LFSCK instances tried to talk with this MDT during the crash for MDT-object verification. When the failed MDT recovers, it will re-start the first-stage scanning from the latest checkpoint. When it finished its local first-stage scanning, it will query other LFSCK instances' status. At that time, the other LFSCK instances will tell it whether it has missed some requests for MDT-object verification. If not, then all the LFSCK instances can proceed normally for the second-stage scanning. Otherwise, the LFSCK instance on the ever failed MDT will skip orphan MDT-object handling in the second-stage scanning as we described above because the LFSCK cannot know the relationship between the skipped name entries and the orphan MDT-objects (especially consider the unmatched name entry and MDT-object pairs). They will be handled when the DNE LFSCK runs next time.

10.2 Recovery from the second-stage scanning failures

If the MDT crashes during the second-stage scanning, before it's restart, other MDTs may need to communicate with the failed MDT for remote name entry verification. Because it cannot respond, those LFSCK instances cannot verify related linkEA entries, so they will mark the LFSCK trace file "`lfsck_namespace`" as `LF_INCOMPLETE` and go ahead to handle other objects. Those skipped objects will be handled when the DNE LFSCK runs next time. It may be fortunate enough that no other LFSCK instances tried to talk with this MDT during the crash for linkEA verification. When the failed MDT recovers, it will restart the scanning from the latest checkpoint. If there are no other failed MDTs, the LFSCK on this MDT can proceed normally without skipping objects.

11 Interoperability and Compatibility

DNE LFSCK is a new feature, and needs the MDT to support it. LFSCK 3 will not run on an MDT which runs an old Lustre release. Since there is no interoperability between DNE MDTs running different Lustre releases, such LFSCK interoperability issues will not affect the usage.

LFSCK 3 reuses the existing user space tools/interfaces to start/stop/query LFSCK, and will extend the user space tools to support more start options. Old users of the Lustre-2.6 release can start the DNE LFSCK with those new options as default mode.

LFSCK 3 extends the on-disk trace file "`lfsck_namespace`", which will not be recognized by old LFSCK. To resolve compatibility issues, we introduce new magic for this file as described above in section 3.
