Clustered Metadata Design

Author	Date	Description of Document Change	Client Approval By	Client Approval Date
Wang Di	04/27/09	Cluster Metadata HLD (version 1)		
WangDi	06/01/09	Cluster Metadata HLD (version 2)		
	06/15/09	JKD - Copyright notice		

1 Introduction

Allowing only a single Metadata Target (MDT) in a filesystem means that Lustre metadata operations can be processed only as quickly as a single server and its backing filesystem can manage. To date, this has not been a critical limitation and it has in the past been addressed by selecting an MDS node and MDT storage that are capable of handling the required metadata load. However, as a Lustre filesystem grows, a single MDS+MDT becomes a performance bottleneck and constraint on the total number of files in the filesystem.

It is not possible to cost-effectively scale a single MDS node to meet the demands of the largest compute systems. Increasing the amount of RAM, number of CPUs, number of network and disk interconnects in a single MDS node increases the cost of the system disproportionately to the amount of incremental performance gained. Allowing Lustre to increase the metadata performance by adding lower-cost independent MDS nodes and independent MDT filesystems ensures that system architects can scale the metadata performance in a linear manner, similar to the way Lustre scales the IO performance by adding OSSes.

The metadata performance limit is being addressed by the addition of Clustered Metadata Server (CMD) functionality to Lustre. With CMD functionality, multiple MDSes provide a single filesystem's namespace jointly, storing the directory and file metadata on a set of MDTs.

Clustered Metadata (CMD) means there are multiple active MDS servers in one Lustre file system, the MDS workload can be shared among several servers, so that the metadata performance will be significantly improved. For the HPCS project, some requirements are hard or even impossible to achieve without CMD. For example one requirement is that the system be able to create 40,000 files per second, which would be difficult to achieve with a single MDS. Clustered Metadata is essential for the HPCS project.

Although CMD will improve the performance and scalability of Lustre, it also brings some difficulties. The most complex one is recovery. In CMD, one metadata operation may need to update several different MDSs. To maintain the consistency of the filesystem, the update must be atomic. If the update on one MDS fails, all other updates must be rolled back to their original states. Unfortunately, the current recovery model does not support this kind of recovery, which can only keeps the metadata operations atomic in a single MDS. This recovery problem will be ultimately fixed by global epochs, which will not be discussed in this design document. Instead this document will propose another way to fix this problem for the short term, and the detail will be discussed in section 4.2. Clustered metadata will be released in two phases. In the first phase, CMD will be released with synchronous distributed operations. In the second phase, global epochs will be brought into the infrastructure and fully support CMD recovery. This document will only discuss CMD in the first phase.

2 Requirements

The requirements in the first phase include:

1. CMD should show MDS scalability compared with a single MDS. The HPCS program has the following objectives for Metadata scalability and performance:

    - A single process must be able to open 100,000 files simultaneously.
    - The file system be able to hold a trillion files.
    - A directory must be able to contain 10 billion files.
    - Able to create 40,000 files per second in the file system.

    These objectives can not be satisfied by the current release of the Lustre system[1](<= 2.0), for example, a single MDS (based on ldiskfs) has an upper limit of 4 billion files per system and 125 million files per directory, which are far less than these objectives. Some of these objectives will be reached by ZFS[2] in Lustre, but requirements such as creating 40,000 files per second and performing an fsck on a Lustre filesystem with 1 trillion files (10^12) in 100 hours, are hard to achieve with single MDS.

2. Any metadata operation that works in a single MDS should also work in a CMD environment.

3. CMD should be compatible with the previous single MDS version. i.e. a file system with a single MDS can be upgraded to CMD smoothly.

4. Although phase 1 recovery will not be fully functional, it will meet the

[1] http://wiki.lustre.org/index.php/Lustre_FAQ
[2] http://arch.lustre.org/index.php?title=Architecture_ZFS_for_Lustre

following requirements;

- The effects of any failure are limited to aborting operations in progress. i.e. no additional damage.
- The namespace remains usable after initial recovery. i.e. no dangling links, disconnected subtrees. There maybe some space leakage after recovery, please refer to section 4.2 for details. And these space leakage will be resolved by online lfsck, which is discussed in another HLD.
- The namespace is restored to full consistency after all recovery has completed. (i.e. fsck cleans up any problems leftover from recovery.

2.1 Definitions

Object: used in the this document to indicate files or directories.

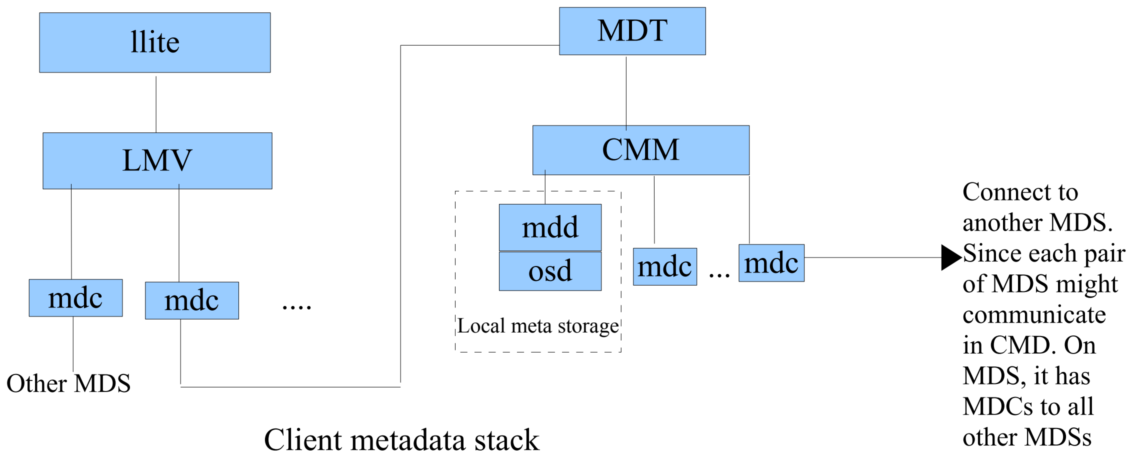
MDT: the storage (filesystem) on an MDS node. There may be multiple MDTs on a single MDS node.

Master/Slave MDT: In CMD, metadata operations might be executed in several MDSs (MDTs). The client first sends a request to one MDT, which is called the Master MDT in this operation (please see FID section in 2.1 about how clients choose a Master MDT for each operation), then the Master MDT distributes the request to other MDTs if necessary, the receiving MDTs are slave MDTs. How master MDTs distribute metadata operations will be explained in section 4.2.

3 Functional specification

3.1 General process

3.1.1 Architecture description


llite
LMV
mdc
mdc
....
Other MDS
MDT
CMM
mdd
osd
Local meta storage
mdc
...
mdc
Connect to another MDS. Since each pair of MDS might communicate in CMD. On MDS, it has MDCs to all other MDSs
Client metadata stack

3.1.2 New modules

The MDS server stack is totally reconstructed for CMD. The MDT[1] will unpack requests and deal with lock and recovery issues. The CMM manages inter-connection between MDSs. MDD and OSD are local modules, MDD is the metadata layer; OSD is the object storage layer, which exports the object API for the MDD, for example create/unlink object, to help MDD finish metadata operations. The LMV is added on the client side to help manage connections between client and other MDSs.

3.1.3 FID

In previous Lustre version(<2.0), because there are is only one active MDS(MDT), a client does not need to know which MDS(MDT) it should access to get the required information for an object, the ldiskfs inode number on the MDT is actually used as the identification for those objects. With CMD, client also needs to know which MDT it should go to for accessing the object, so the MDT inode number is not enough for the identification any more. FID is introduced into Lustre in 2.0 to resolve this problem. A FID is a three item data structure,

[1]This MDT does not mean the filesystem. It is actually one of the service level. Confusing name.

{f_seq, f_oid, f_ver}, in which f_seq is 64 bit, f_oid is 32 bit, f_ver is 32 bit, and it is a cluster-wide unique identifier for a file or a directory. A FID can actually be used everywhere in Lustre, for example files/directories on MDT, data objects on OST and llog objects on MDT etc. But currently FIDs is only used to identify files and directories on MDT.

When a client tries to locate an object by the FID, it first looks the object up in a FID location database (FLD) to get the MDT index which indicates on which MDT the object resides, and then gets the object by looking it up in the object index table on the appropriate MDT.

1. The FLD used in the first lookup is indexed by the FID sequence number(f_seq), i.e. the objects with the same sequence number will reside in the same MDT. The FLD is stored in the root MDT[1][2], but each node has its own local cache. If it can not get an MDT location from the local cache, it goes to the root MDT, and gets the location information there and updates the local cache at the same time. Note: once an object is created, it will not move to another MDT, meaning its FID entry will not be changed during the life of an object. This also makes the implementation of FLD cache easy. The object index table used in the second lookup is for mapping the FID to the object, and it is currently implemented by the ldiskfs htree structure, but with a higher tree level.

With FIDs, when creating a new object, the client chooses an MDT and asks it to allocate a FID for the object, then do the following creation with the FID. For efficiency, a client usually pre-allocates some FIDs and caches them locally, so it does not need to request FIDs from servers every time. When recovery is occuring between clients and servers, these unused, pre-allocated FIDs in cache will be thrown away and clients will request new FIDs from servers, i.e. the server will not recovery these preallocated FIDs during failover, which is different with the preallocated id between an MDS and OST.

3.2 Scalability

3.2.1 Single client operation

Even though there are multiple metadata servers, metadata operations are still handled in partially serialized fashion, i.e. the client only sends the request to one MDT each time. And furthermore, a master MDT may someimes need to send

[1]Just as with OSTs, each MDS has an index in CMD. The MDS with 0 index is called root MDS.
[2]The FLD may be mirrored across MDTs in a future release.

requests to other MDTs. Compared with single MDS, metadata operations in CMD need more RPCs and disk operations. But, in certain scenarios, for example creating large amount of files , multiple severs can share te workload, and clients can expect quicker responses from MDTs.

3.2.2 Multiple client operations.

The throughput of metadata operation should be improved for multiple client operations.

3.2.3 Multiple client access same directory

With CMD, directories also have layout information (setting stripe), similar to files. Multi-stripe directories will be split into several parts and each of them will reside on different MDTs. If multiple clients access this directory at the same time, different clients can access different MDSs (MDTs), and the server load will be shared.

3.2.4 Multiple client access different directory

When multiple clients access different directories at the same time, these directories are usually on different MDTs, clients access different servers, and the server load will also be shared by these MDSs(MDTs).

3.2.5 MDS load imbalance

Currently the directory layout information can not be changed dynamically. If a large number of clients access one directory at the same time, or access the same part of the directory, the load on that MDS might be quite high. This could be resolved by re-stripeing the directory dynamically, i.e. the entries of that directory can be dynamically relocated to other MDSs, but it depends on the directory layout lock, which will not be implemented in this project. Dynamic restreing of directories will not be supported in the initial release of CMD.

3.3 Recovery

The long term recovery solution for clustered metadata recovery will be global epochs, which will not released in phase 1. Instead those metadata operations that span multiple MDSs(MDTs) will be synchronous to avoid recovery once the system crashes, this might impact the metadata performance. The detailed algorithms will be discussed in section 5.2.

3.4 Compatibility

This section addresses compatibility and differences betweena file system with a single MDS and one using CMD.

3.4.1 Upgrading to CMD

When upgrading from a single MDS (MDT) to CMD, the required steps will only be to: mkfs the new MDTs and mount the new MDTs. Clients will be notified that the new MDS(MDT) have been added to the system, and will start to create new objects on these MDTs. The addition of MDTs to a CMD file system is handled the same way.

Because in older versions of Lustre (prior to 2.0) directories do not have stripe information, after upgrading to CMD, the stripe count of the existing directories will be 1.

3.4.2 Disk format compatibility

In 2.0, with a single MDS, the FID is actually stored in the inode EA. A client retrieves the FID from the EA after it gets the object. But for CMD, the FID must be stored in the name entry, in this case the name entry and inodes may not be in the same MDT(cross-ref object). CMD might either retrieve the FID from the EA(for old objects) or from the name entry(for newly created objects for CMD).

4 Use cases

4.1 Metadata operations

For clients, the metadata operations remain the same as for a single MDS. But related tools will be provided or changed for CMD.

1. lfs find show the file location information, including which MDT the file resides on.

2. lfs dirstripe can be used to set directory stripe information.

4.2 Scalability

An HPCS project goal is to be able to place 10 billion files in one directory. These files should be put into a multi-stripe directory. Suppose the stripe_count is N, then the directory entries will be distributed over N MDSs(MDTs). HPCS also desires that the system be able to create 40000 files per second, which can only be satisfied with a multi-stripe directory.

4.3 Recovery

To facilitate recovery, in the initial releases of CMD, the cross-MDT operations will be done synchronously. During a recovery, when the master MDT receives the resend and replay requests from the client it only checks the status locally and determines whether it needs to redo the requests. The Phase 1 recovery mechanism will at worst leak a small amount of space after an MDS failure.

5 Logic implementation

5.1 Scalability

5.1.1 Namespace displacement

To achieve scalability of CMD, inodes (files and directories) should be spread across all MDSs(MDTs) as evenly as possible. The Phase one placement policy is to only distribute directories, i.e. files are always on the same MDT as their parents. For a new directory, the MDT will be chosen based on a hash of the directory name being created. The available space of each MDT is also a factor. The rules are:

1. The MDT, whose available space is largest, will be preferred.

2. If the available space is similar for each MDT, it will choose an MDT using the following rule:

   a. If it is a single-stripe (non-striped) directory.

      MDT_idx = (summation of all the character's value) mod (MDT_count).

   b. If it is a multi-stripe (striped directory, see 4.1.2) directory.

      The inode will be placed in the same MDT as the name entry.

These are the default striping rules. The user can define alternative policy to specify where to put the inode instead of using the above policy. For example users can set a policy that new child directories stay on the same MDT as their parent.

Since the directory is placed on a new MDT at the time of creation, it might be in a different MDT than its name-entry, this is called across-ref directory. Note: because of this separation, it needs two RPCs to lookup a cross-ref directory.

1. Go to the name entry MDT(Master MDT), get the FID.

2. Go to the object MDT(slave MDT), get the real attributes using the FID.

We can see that there are some performance penalties to accessing a cross-ref directory compared to a normal directory.

5.1.2 Striped directory

Another way to distribute the directory is to set the stripe information for the directory, which will split the directory across multiple MDTs.

Stripe information for a directory is similar to file stripe information, but there is no stripe_size. Only a stripe_count and stripe_index. The whole name space of the stripe directory will be split into several sections according to hash value. For example, if the stripe count of the directory is N and hash range is 0 to MAX_HASH, then first MDT will keep records with hashes [0 ... MAX_HASH / N - 1], second MDT with hashes [MAX_HASH / N ... 2 * MAX_HASH / N – 1] and so on. The hash value of each entry is computed with the Tea hash algorithm[1] and the input cookie is the name of the entry. Note: a different directory may have a different hash algorithm, so the hash function will be part of the directory stripe information.

Ideally, the directory should be re-striped dynamically, for example, when a single -stripe directory size reaches an upper limit, it can be re-striped to a multiple stripe directory, but this depends on the directory layout lock. Dynamic directory re-striping will not be implemented in the initial phase of CMD.

5.2 Metadata operations

[1] http://en.wikipedia.org/wiki/Tiny_Encryption_Algorithm Given weakness of the TEA hash, we might consider some other algorithm like md4.

Metadata operations in CMD are quite different than metadata operations in a system with a single MDS. The three main differences are:

1. The client needs to choose the right MDT to handle the requests by checking the FLD.

2. A singel operation might involve more than one MDT. The Master MDT might send the operation to a slave MDT. As we discussed in section 3.3, this kind of request between MDTs will be ordered and partially synchronous to ensure that in the case of multi-node failures at worst a single reference on the inode will be leaked. This might lead to some files not releasing, but the namespace is usable after the recovery.

3. During the recovery process, when the master MDT gets replay or resend requests from clients, it will first check the local status to know whether this operation has been completed or not. If it has not been, the master MDT will redo the operation. Although the RPC between master and slave is synchronous, the master MDT does not know whether the resend(not replay) request has been done or not in theslave MDT, because the master MDT might crash after it sent out the RPC to the slave MDT. This can be resolved by adding the request xid (from client to the master) to the last_rcvd file. When master MDT sends the RPC to slave MDT, the xid of the request (from client to master) will be included in the RPC, and the slave MDT will add the xid to the corresponding entry of last_rcvd file. Then the slave MDT will know whether the request(from the client) has been processed or not during the recovery.

The following describes the problems(leaked inodes) that might occur with this simple synchronized recovery and how they are addressed:

5.2.1 Open/Create

1. The client chooses one MDT and allocates a FID for the new file. If the parent is a multi-stripe directory, it will choose an MDT according to the rule discussed in 5.1.2. If the parent is a single-stripe directory, the file will be created in the same MDT as its parent.

2. The client sends the open/create request to the master MDT. If the parent is a multi-stripe directory, the master MDT is determined using the rule in 5.1.2. If it is not, the parent directory resides on the master MDT. Then, in the master MDT, the name entry will be inserted into the directory and the file will be created. Because all updates are occurring on the Master MDT, the replay and resend

request handling will be the same as with a single MDT.

5.2.2 FID update

Another important topic regarding create is FLD consistency. Once the FID is allocated, and its object is created and committed to disk, the system must make sure the corresponding FID location information is also stable. As discussed in section 3.1, there are two kinds of FID location information.

1. The object index is used for mapping the FID to the real object inside the MDT. Its update will be included in the same transaction as the object create. So its update will be synchronous with the object create.

2. The FLD is used for mapping the FID to the MDT index. The FLD is updated in the root MDT when the client pre-allocates the FID. And the update is outside of the transaction of the following create, even on a different MDT. So the FLD updates will be a synchronous operation. However, since the FLD is expected to change very rarely, this will not have a noticable performance impact.

5.2.3 Mkdir

For mkdir the following occurs:

1. The client allocates a FID for the object. If the parent is not a multi-stripe directory, it will choose an MDT according to the rule discussed in 5.1.1. If the parent is multi-stripe directory, it will choose an MDT using the rule in 5.1.2.

2. The client sends the mkdir request to the master MDT, where the parent resides. The master MDT checks whether the FID is local or not. If it is not, the master MDT sends the create request to the slave MDT and creates the object synchronously. After that, it will insert the name entry for the object into the directory. Two notes here,

   a) The create request between the Master MDT and the slave MDT is synchronous,

   b) The name entry insertion update needs to be committed to disk before any further creation under the new directory in case these creations might include any synchronization operations.

When the server receives resend mkdir requests, it will check whether the name entry exists. If it does, then the server does not need to do any updates, since the synchronize process will make sure the object on the slave MDT exists. If it does not, it will try to recreate the object on the MDT. On a slave MDT it will know whether the request has been done by checking last_rcvd.

For areplay request, it will first check whether the name entry exists, if the source entry (A) is still in dir_A, the server will only insert the name entry then return, since the object must exists because of the synchronization between master MDT and slave MDT. If the entry exists, the server will do nothing and return.

5.2.4 Link

For Link operations:

1. The client sends link requests to the Master MDT. If the parent is a multi-stripe directory, it chooses the master MDT by the name hash value according to 5.1.2. If the parent is not a multi-stripe directory, the master MDT is the MDT where the parent resides.

2. The master MDT will check whether the FID is local or not. If it is not, the master MDT will send the request to the slave MDT to increase the reference count of the object, which is synchronous. Then insert the name entry of object.

When the server receieves resend link requests, it will check whether the name exists. If it does, the server does not need do any updates for the operation. If it does not, it will redo the link. On a slave MDT, it will know whether the request has been done by checking last_rcvd.

For a replay request, just as with mkdir, it will know whether the request has been processed by checking by the name entry in the master MDT.

5.2.5 Unlink

For unlink operations:

1.The client sends unlink requests to the Master MDT. If the parent is a mutli-stripe directory, it chooses the master MDT by the name hash value according to

5.1.2. If it is not, the master MDT is the MDT where the parent resides.

2.If the object is local, the master MDT can do an asynchronous unlink just as occurs with a single MDS. If it is a remote object, the Master MDT first deletes the name entry synchronously, then decreasesw the refcount of the object on slave MDT asynchronously.

When the server receives resend unlink requests, it will check whether the entry exists. If it does, the server will redo the unlink. If it does not, it will not do any updates and return. On a slave MDT, it will know whether the request has been done by checking last_rcvd.

5.2.6 Rename

Since rename will involve more objects (four objects) and steps, and those objects can be anywhere over the cluster, rename is more complicated than other operations. Rename actually includes two phases: link and unlink. Basically, the link process is synchronous, while the unlink process is asynchronous. Since rename for a directory and a file are quite different, we will describe them separately. To describe things simply, the following description will not mention whether the operations should be done locally or remote. If it is remote (not in the master MDT), it means the request need to be sent to one of its slave MDTs.

5.2.6.1 Rename regular file (rename dir_A/A dir_B/B)

MDT1(master MDT) holds dir_A. MDT2 holds A, MDT3 holds dir_B, MDT4 holds B.

1. Client sends rename requests to the master MDT1
2. MDT1 increases the refcount of A synchronously.
3. MDT3 replaces entry B with entry A synchronously. If B does not exist, just add entry A.
4. MDT1 deletes entry A asynchronously.
5. MDT4 decreases the refcount of B asynchronously, if B exists.
6. MDT2 decreases the refcount of A asynchronously.

5.2.6.2 Rename directory (rename dir_A/A dir_B/B)

MDT1(master MDT) holds dir_A. MDT2 holds A, MDT3 holds dir_B, MDT4 holds B.

1. Client sends rename requests to the master MDT1.
2. If B exist, MDT4 checks whether there are entries under B, if there are, rename failed(ENOEMPTY), if there are not, MDT4 increases the link count of B synchronously.
3. MDT2 increases the refcount of A synchronously.
4. MDT3 replaces entry B with entry A synchronously. If B does not exist, just add entry A.
5. MDT1 deletes entry A and decrease the link_count of dir_A asynchronously.
6. MDT2 decrease the refcount of A asynchronously.
7. MDT4 decrease the refcount of B asynchronously.

So basically, rename can be divided into two steps, link and unlink. The link step is the key to making the filesystem usable , and it must be synchronous in the current recovery model, while the unlink step can be asynchronous, and the failure of this step will only cause some leaked inodes. When the server receives a resend request, it will check whether the source entry(A) exists under dir_B. If it does not, it will redo the entire rename processes. So, the link process might happen twice, which will cause:

1. For regular file rename, the refcount of A is increased twice, and it becomes an orphan object.
2. For directory rename, the refcount of A is increased twice, the link_count of B is increased twice. So A and dir_B become orphan objects when all of the directory entries are removed.

If it does, it will do nothing and return. So, if the unlink (asynchronously) failed, then

1. The refcount of A can not decreased. A and B will refer to the same object.

2. The link_count of dir_A might not decreased. dir_A becomes anorphan object.

During the replay process, the server does not need to do anything, because replay request indicates the synchronous step s (step 1, 2,3,4) has been executed on the server. But since it does not check whether the unlink steps succeeded, some leaked inodes might be left after recovery. Note: Because the rename process is quite complicated compared with other operations we may not try to enhance recovery as we did for other operations (checking last_rcvd on slave MDT), so some leaked inodes might be left after failover.

5.2.7 Rename check

Rename check is used to avoid rename recursion. i.e. the source can not be an ancestor of the target, which is not required if the source and target are regular files. The checking is actually implemented by finding the common parent of the source and target, which is also needed in the following rename lock process. If the common parent is not the source, it means the source is the parent of the target, so is_subdir check succeed, otherwise it is failed. We can always find the parent by lookup .. entry.

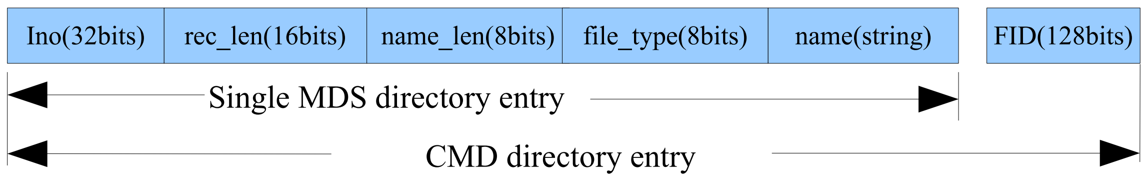
5.2.8 Rename lock

In the CMD implementation, the rename check process will be protected by **a single global lock** which will serialize all cross-directory renames to ensure the rename check is valid. This global rename lock is only used to protect the rename check process, and it is actually not needed after it acquires all of the rename objects lock (ldlm lock), so it can be dropped during the actual rename process. Note: this global rename lock does impact rename performance, and we considered implementing an FS hierarchy lock, instead of a global lock. But it requires much more complicated lock mechanisms. Since rename is usually a rare operation iwe chose to use global rename lock.

5.3 Compatibility

To upgrade a file system with a single MDS to CMD smoothly, CMD also needs to understand the disk format of single MDS(MDT) systems, so it will not be necessary to reformat the filesystem when upgrading from a single MDS to CMD.

The major difference between the single MDS disk format and CMD disk format is the directory entry. As discussed in section 3, CMD will store the FID in the name entry. Single MDSs store the FID in the inode EA.



The FID will be appended after the name. Name_len will be extended and the name will be NUL terminated to include the FID and a new file_type will be added to indicate there is a FID after the name.

6 State management

6.1 Scalability & performance

While there is some overhead in CMD compared to a single active metadata server, the large majority of metadata operations such as file creates, attribute lookups, and unlinks will be performed by the single MDS that manages a particular inode. Performance for most individual operations is comparable to a system with a single MDS, but the individual operations can be performed in parallel on multiple MDSes so the aggregate metadata performance is expected to scale in a nearly linear manner with the number of MDSes in the filesystem.

A limited number of less frequently performed operations, such as some sub-directory creation, cross-MDT rename, and hard-link operations may involve multiple metadata servers, and will need multiple RPCs to complete. The number of cross-MDT operations will be actively limited by file creation policy specified by the user and/or administrator in a similar manner to the striping of Lustre files today. Individual directories can be split over multiple MDTs to improve aggregate performance within that directory, but even for split directories the file creation, lookup, and unlink for specific filenames will normally be considered local MDT operations and will be completely asynchronous.

6.2 Recovery changes

The recovery changes and consequences has been discussed in section 5.2.

6.3 Locking changes

This sections describes the changes to locking with CMD.

6.3.1 Cross-ref object locking

For a cross-ref object, the name entry and the object are in different MDTs. The name entry is in the master MDT, the object is in a slave MDT. When performing a lookup of a cross-ref object, it gets a lookup lock from the master MDT and gets an update lock from the slave MDT. For example, when you do

“ls -l a/b” a is in MDT1, b is in MDT2.

1. Client will get b's lookup lock from MDT1, because b's entry is on MDT1.

2. Client will get b's update lock from MDT2, because b is on MDT2.

Compared with a single MDT, CMD also separates the lock of a single object, which also brings some differences.

In a single MDT, the namespace and permissions (uid/gid/acl) are protected by a lookup lock, and all other inode attributes are protected by an update lock. If some one changes the inode's permissions, MDS will revoke lthe ookup lock of the clients, and then clients need to re-acquire the permission from the server. But with CMD, the permission changes happens in the slave MDS(MDT) where the object resides, but it only owns the update lock, and the lookup lock can not be revoked at this time. So in CMD, the permission lock will be split from the lookup lock and become a new inode bit lock. The slave MDS(MDT) will own the permission lock, and it can revoke the permission lock on a client when necessary.

6.3.3 stripe directory locking

For a multi-stripe directory, the master MDT is the MDT which its FID refers to, the slave MDTs are the MDTs which it will be split across, except the master MDT. The master MDT owns the lookup lock, and the update lock is owned by all the master and slave MDTs.

The inode attributes of a multi-stripe directory will be protected by the update lock on the master MDT. But the entries of a multi-stripe directory will be protected by the update lock on all master and slave MDTs, and each MDT will only own the lock of those entries residing on that MDT. So when a client reads entries, it will get all the update locks from the MDT where those entries reside. When the MDT revokes the update lock from the client, it will also only let the client invalidate those entries residing on that MDT, instead of revalidate all of them. But when unlinking the multi-stripe directory, it needs to get all the update locks from the master and slave MDTs.