Rollback HLD

Peter Braam, Mike Pershin

February 9, 2008

1 Engineering Requirements

Rollback is a recovery mechanism for a cluster of metadata servers (**CMD**). If the system crashes due to a power failure or due to multiple MDS failures, the problem that we face is that the state of the cluster may not represent a valid file system. The reason it may not is that transactions on different nodes may be related to a single operation at the file system level. Some of these transactions may be lost in the crash, others may have committed to disk.

In order to address this problem, the nodes will engage in a distributed algorithm that restores the disk state to a snapshot which we will call an **image snapshot** (2.1.3). Nodes will rollback to the snapshot, based on an **undo log** (2.1.2).

The requirements for this component are:

1. Define the image snapshots
2. When a cluster failure takes place, calculate operations to undo on each node, so that a consistent snapshot is reached.
3. During normal operation, discard of the portion of the undo-operation log that is not needed for undo in case of recovery.

2 Functional Specification

The required functionality can be achieved by introducing two major component - **undo log** and **image snapshot**.

2.1 Definitions

2.1.1 Types of operations

fsop - an operation on the initiating MDS which received client's request.

depop - The initiating metadata server may involve another metadata server in the process: this other metadata server executes a dependent operation which we call **a depop.**

crossop - filesystem operation that consist of several small operations on different MD servers.

2.1.2 Undo log

Each MDS node creates and constantly updates log of all operations for each **fsop** or **depop.** The key requirements to such log are:

1. Log contains enough information to replay back logged operation returning node to the some recent state;
2. new record should be written to the log before **fsop**/**depop** will affect the system or at least be in the same transaction;
3. log should be able to add new records, delete unneeded and iterate over logged records forward and backward.
4. log should contains information about latest committed epoch.

Actually *llog* functionality is well suitable for these requirement, so below in document we will use *llog* terminology while describing the undo log. Nevertheless the different approach may be used also, but the choosing such functionality is out of scope of this document.

2.1.3 Image snapshots

The request on MD server can invoke the additional request on another server. In two cases (rename and directory split) more nodes or repeated transactions can be involved, and we build a stack of initiating and dependent calls: each node starts at most one **depop** (depended transaction even) on another node.

Each metadata server executes local memory transactions which follow a **start/stop** pattern. The memory transactions are collected into a disk transaction which sees **open/commit** operations. The disk transactions on each node are strictly ordered. If transaction A is started before B, then A will belong to a disk transaction equal or earlier than B. Each memory transaction has a Lustre transaction number and a corresponding **undo** record in an log, which is transactionally maintained.

File system operations (**fsops**) have dependencies, e.g. a file in a directory that doesn't exist yet cannot be created. The dependencies are determined by the read and write set of the memory transactions associated with the transactions. If two transactions have an overlap in this read-write set, they are called dependent. On a single node, the start and stop events of dependent transactions are serialized by the file system. By using locks and because start order is preserved in disk transactions the disk images of file systems on single nodes are consistent wrt the **fsop** dependencies.

Definition: A **image snapshot** of a clustered metadata file system is a disk image that has the following properties:

1. For all file system operations P depending on Q: if the effect of P is in the image, so is that of Q.
2. For all **fsops** which involve dependencies among the metadata server transactions: If the initiating transaction of a file system operation is in the image, so are all its dependencies.

2.1.4 Epoch

A transaction on a node belonging to a snapshot and not to a previous snapshot on that node is said to lie in the **epoch** of that snapshot. Clearly an image snapshot is a file system that is reachable by file system operations and all transactions and their dependency stack are completely incorporated. Epoch is a part of record in undo log. So we can control the committed epoch and find the needed records in undo log.

2.2 Rollback algorithm

After definitions of **undo log** and **image snapshot** we can describe the **rollback** algorithm. After failure a cluster can rollback to the last committed snapshot using the followed algorithm:

1. scans the **undo log** backward on each MD server and find the latest committed epoch;
2. choose the minimum latest committed epoch among all servers;
3. rollback to the snapshot with chosen epoch by doing undo operation in accordance with undo log;

Now server is in consistent state and recovery may starts.

2.3 Snapshot functionality

2.3.1 General approach

The proposed fault-tolerance functionality is based on snapshot. We need to write snapshot frequently to nullify the long restoring after failure. There is the coordinating node, called the coordinator which takes care about snapshot state and should be able to do the following:

- initiates the epoch increase (start new snapshot) and sends RPC to other nodes;
- receives nodes states (last committed epoch, current epoch, etc.) and calculate collective state;
- initiate the purging of undo log for committed snapshot (end of the snapshot).

Each node in a cluster is the coordinator for corresponding snapshots, e.g. snapshot P is controlled by node N = P % K, where K is total number of MDS.

Lemma 1. *The difference between epoch on cluster nodes cannot be more than 1.*

Proof: As snapshot coordination is doing via network, there is a gap between epoch increase on different nodes. Suppose that the old epoch was N. So two cases are possible:

1. Node already get message and its epoch is $N + 1$;
2. node didn't receives the control message and its epoch is still N

Considering that there is no concurrent epoch increase the epoch difference between node cannot be bigger than 1.

2.3.2 Epoch control

The control increase is done periodically by control node. The control increase starts the new snapshot with new epoch.

Each server does the following:

1. takes new epoch from RPC
2. if the local epoch should be increased then
   (a) update the local epoch
   (b) all new crossop or depop should wait until the epoch increase will be done.
3. reply to the control node
4. After the epoch increase all new operations will use new epoch;

2.3.3 Transaction control

Each MDS should identify the local last committed epoch and the cluster-wide last committed epoch - the biggest epoch which is committed on all nodes.

Local last committed epoch should be used to determine the cluster-wide value by choosing the minimum value.

Cluster last committed epoch should be reported to the clients so they can drop all pending operations for that epoch. The MDS uses this value to cancel records in undo log.

So transaction control functionality should care about two tasks:

1. Negotiation between MDS about cluster-wide last_committed_epoch;
2. Determine the local last_committed_epoch value.

With the response to a control message, received possibly asynchronously by the coordinator, the last completed and committed epoch on each node can be reported. The coordinator will send a second message requesting purging of unneeded undo records immediately after its knows the collective answer from all nodes, but the coordinator only does this if it has moved since the last purge. This message also indicates to other nodes that the snapshot has completed and the next MDS node can become the leader for the next snapshot.

In case of recovery the last committed and completed epoch are again collected and all nodes again roll back to the end of the last epoch committed on all nodes.

2.3.4 Cross-ref operations

The multi-node operations should be at the same epoch - it is definition of a snapshot. But we can encounter the situation when part of such operations can be in different epochs due to **Lemma 1.** This situation need to be handled correctly and all depops should be in the same epoch with fsop.

The situation with different epoch number on depended servers can occur during epoch increasing procedure when some servers did already epoch increase and other - not yet. Due to the **Lemma 1** here is two possible situations:

1. Initial epoch on first server is already increased, but epoch on remote server is old. In this case remote server should update epoch and continue with the current operation.
2. Initial epoch is less than one on the remote server. In this case the remote server complete operation in new epoch and reply with new epoch, originated MDS gets answer and update the epoch before continue with local operation.

The MDS initiating the depended request does local operation only after all depended remote calls therefore epoch will be negotiated before starting of local operation. There shouldn't be concurrent epoch increase while multi-node operation is in progress so some protection is needed for this.

2.4 Affected functionality and API

2.4.1 API changes

Rollback needs several new methods and structures in MDT and CMM to provide the epoch control. For writing the epoch record into log the existent llog API can be used in CMM/MDD/OSD.

All rollback functionality is implemented in CMM layer with help from MDT/MDC for epoch control and from MDD/OSD for undo logging.

2.4.2 Recovery

Client should use the epoch for all pending operations instead of transaction number. MDS will provides the information about last committed epoch to the clients. This should be the epoch that is already committed on all MD servers. After that the client will drop pending requests with that epoch.

2.5 Logging operations

Good place should be chosen for intercepting operations and transferring them to the undo log. There are several possible ways to do that:

- OSD level can do the log all local fs operations
- Interception in CMM level which can log all operations

3 Use Cases

3.1 Directory distribution

There is special type of file or directory in CMD - cross-ref object. Directory entry for that object is placed on one MDS but object itself - on another MDS. Therefore any operation with such object via its name will go through MDS where its name is stored to the MDS with object itself.

fsop - operations with name

depop - operations with object that is placed on remote MDS

3.2 Link

Link operation can use file on another MDS while creating a link on the local on.

fsop - creating the new entry in directory

depop - increasing link counter in object on another MDS

3.3 Rename

As rename operation can change name and even parent, there are several cases which will produce **depops.**

3.4 Directory split

This case is the most hardest one. Directory split can produce a lot of **depops** and can be very long in time. Moreover design of split functionality is not finished yet. In this document we will suppose that splitting produces reasonable number of **depops**.

4 Logic Specification

4.1 Details of epoch control

Each snapshot is a structure which contains the epoch number and ref-counter. While starting in that epoch the operation will take the reference to it. The reference is dropped when related transaction will be closed in memory.

4.1.1 Control increase of the epoch

The control node set new epoch locally and send RPC (possibly asynchronously) to the other nodes.

The other nodes receives the increase request and do the following:

1. check if local node is already in new epoch (this can be if some depop was received before the control message)
2. update local epoch if needed
3. do reply

4.1.2 Determining the global last committed transaction

Let's suppose that all nodes know their local last_committed epoch. To determine the global value the coordinator node do the following:

1. request the last committed epoch from all nodes;
2. calculate the collective state. If this value is not changed then exit;

3. send the result to all nodes. Nodes can purge the undo log and update last_committed_epoch value for clients.
4. coordinator node send control message to the next coordinator and it will start new epoch.

It is possible that determining the last committed epoch can be partially done while control increase negotiation, e.g. nodes can report local last_committed_epoch during reply to the epoch increase message.

4.1.3 Cross-ref operations logic

The multi-node operations should be at the same epoch - it is definition of a snapshot. But we can encounter the situation when part of such operations can be in different epochs due to **Lemma 1.**

This case can be handled by following way:

1. get the reference on the epoch
2. send depop RPC to other MDS and get reply with epoch value in RPC
3. check the epoch returned in reply:
   (a) If it is bigger than current one then set new epoch locally;
   (b) drop reference to the old epoch;
   (c) take reference to the new one;
4. do local operations with in the referenced epoch as parameter;
5. put the reference.

The remote server:

1. receives the request for depop, check the epoch
2. if local epoch bigger than one in RPC then update the epoch in RPC;
3. if local epoch lesser than one in RPC then
   (a) set new local epoch;
   (b) drop reference to the old epoch;
   (c) take reference to the new one;
4. do another remote request if needed with the updated epoch;
5. get reply and update local epoch if needed;
6. do local operation with the negotiated epoch;
7. put the reference;
8. reply to the initial server also with the same epoch;

This algorithm will place fsop and all depops in one epoch. Possibly there is a lock is needed when epoch is updated.

When a server starts new operation it takes a reference on the current epoch. Taking this reference is atomic with respect to increasing the epoch, and takes into account that a message received from a remote node to start a dependent transaction may have just increased the epoch.

While the transaction and dependencies are being negotiated with other servers, the epoch may move to a later snapshot. The reference is dropped when the transaction is closed in memory.

4.2 Transactions and undo records

4.2.1 Finishing the epoch

Hence references can only be taken on epoch equal or higher than the current one, older epochs will not get new references, and we merely have to let the commits drain to discover that the epoch has committed. But note that epochs may commit out of order, and an epoch as a whole has only committed if all previous epochs have committed and the transactions in the epoch have committed.

If the ref-count of the epoch is 0 and the current epoch is bigger, the server put the special record (EPOCH_END) into the undo log so last committed transaction can be easily found by scanning the undo log backward for such record. Commit callback for this transaction will means that all transactions from that epoch are committed and local last_committed_epoch can be updated.

4.2.2 Purging the undo log

While control node decides that epoch N should be purged the undo log is scanned forward and all records with epoch N are cancelled until the special records EPOCH_END will be encountered.

4.2.3 Writing the undo log

Undo log is handled by CMM. The CMM adds new record into log before the local operation itself. In that case there is no need to export transaction API to the CMM. The undo record and operation itself can be in different transactions, so there is possible situation when we will have a record in undo log, but no on-disk changes related to the operation. In that case attempt to do revert changes will fail and should be ignored. It is allowed only for last record in the undo log actually

Records in undo log are added/cancelled/scanned and undo log is initialized/closed with help of OBD llog API.

4.3 Rollback

4.3.1 Client Recovery and Image Snapshots

Traditionally clients replay un-committed transactions, these are communicated to clients by metadata servers through last committed numbers. Client recovery interacts with snapshots by starting replay after the rollback has completed. Now the epoch number should be used instead of transaction.

The MDS nodes collectively determine the last committed snapshot. Clients retain all transactions with epoch numbers that are beyond the last committed one, and free those before.

During replay on the server, the server scans the bitmaps in the undo log to determine if a transaction offered for replay by the client requires replay.

4.3.2 Finding the latest committed epoch

Each MDS have to determine last committed epoch for recovery purposes. To do so it scans undo log backward for first EPOCH_END record. The epoch stored in it is a last committed epoch on current MDS.

4.3.3 Avoiding unnecessary undo records

As we explained in the introduction, local file systems roll back conveniently to a consistent state. With good choices of metadata placement, there will be many transactions that are local to a particular MDS. The question is under what circumstances we can avoid writing undo records for such transactions.

For example we can skip undo logging if there were no crossops or depops in current epoch. Maybe some dependency tracking mechanism is possible also, so this should be investigated during DLD phase more closely.

4.3.4 Encouraging early commit

If usage indicates that fsops that involve dependent operations are quite rare then it may be beneficial to immediately:

1. begin a new snapshot
2. nodes involved in the distributed transaction begin to commit the previous epoch
3. nodes can stop recording undo information for certain transactions (see above) when the global commit of this epoch is confirmed.

4.4 Recovery

When a cluster goes into recovery the metadata server with index i = 1 is responsible to gather the current and last committed snapshot from all nodes.

The process begins similarly to the 3 steps discussed in the previous message.

1. Node i = 1 connects to all other metadata servers and enquires about existing exports for the targets. If no target suffered transaction rollback due to a restart, no undo is necessary. Resending will undo the damage.
2. If merely one target failed and the clients and other servers stayed up, no rollback is necessary, replay will fix the problems.
3. During this enquiry node i = 1 requests status from all nodes and computes the globally last committed snapshot.

4. It sends a message to indicate to what point servers should rollback.
5. When this completes replies are sent to the node i = 1.
6. When all messages have arrived the coordinator sends a message to all nodes indicating the rollback is complete.
7. Nodes proceed to accept replay and resent messages.

5 State management

5.1 State invariants

- epoch change is atomic
- crossops belongs to the one epoch
- cluster-wide last_committed_epoch is used for handling pending commands on clients

5.2 Scalability & performance

- For scalability purposes the epoch increase and last_committed_epoch negotiations can be done in parallel where it is possible
- During rollback the optimisation is possible to avoid unneeded extra work

5.3 Recovery changes

- client changes to dropping the pending request is needed;
- server may decide that some replays are not needed;
- rollback should be done before starting the recovery process
- while replay the crossops will call the depended operations also with REPLAY flag

6 Issues for inspection

1. It may be advantageous to record the snapshot right at the end of the disk transaction. It would be worth puzzling about the generic rollback behavior.