1 Rollback - HLD

1.1 Engineering Requirements

Rollback is a recovery mechanism for a cluster of metadata servers. 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**. Nodes will rollback to the snapshot, based on an undo log.

The requirements for this component are:

1. Define the image snapshots
2. When a cluster failure takes place, calculate what 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.

1.2 Definitions & Specifications

1.2.1 Image snapshots

Clients contact a metadata server to initiate the execution of a file system operation, this leads to a an operation on the initiating MDS itself which we call an **fsop.** 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.** 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 on another node.

Each metadata server executes 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 llog, which is transactionally maintained. Transaction numbers and undo records commit in an ordered fashion to disk wrt to the order introduced by the transaction number.

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 transactins: If the initiating transaction of a file system operation is in the image, so are all its dependencies.

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.

1.3 Logic of transactions and undo records

1.3.1 Transaction organization

The initiator will begin a file system operation by taking sufficiently many ordered DLM locks to prevent dependent transactions from starting until the locks are released.

The node then reads its current epoch. It sends a request to a node required for a depop including the epoch number, in an order determined by the nature of the file system operation.

The node receiving the request also takes locks to prevent dependent operations from starting. It then looks at the epoch in the request and if it is larger than its current epoch forwards the epoch locally. It recursively contacts other nodes for depops.

When a node receives a reply from nodes which again includes an epoch, and possibly moves the epoch number for this transaction forward to the epoch received. But it does not take into account any increase of the epoch on the since sending the request for the dependent operation. Finally this node in turn replies to the initiator with the epoch number. In this way, the highest epoch number is chosen along the line up and down the dependent operations.

Now all nodes have agreed on the same epoch number and each can start and complete a disk transaction in that epoch. Finally they release locks.

The initiator replies to the client.

Note that the epoch can increase and a transaction may be started in the later epoch on a node, before a transaction is start which the node had agreed earlier to execute in an earlier epoch.

So when a new epoch is started it is true that no transactions from that epoch have executed already, but transactions belonging to a previous epoch may still be running. So the first transaction in a new epoch is easily identified when scanning backward, but not the last.

1.3.2 Identifying the last committed snapshot locally

When a server plans to get involved in a transaction, 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 reference may move to a later snapshot. The reference is dropped when the transaction is closed in memory.

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 refcount of the epoch is 0 and the current epoch is bigger, the server records the last transaction closed in the epoch, both on disk and in memory; the record on disk should be a consequence of the commit of the transaction as with the last transno. The commit callback of this transaction, together with the and the commit callback of that transaction is an indication that the transactions in that epoch have committed.

1.3.3 Rollback

A cluster can rollback to the last committed snapshot. To do so, it scans the undo log in backward order until encountering the first transaction in the new epoch. Notice that during this scan, some records might be encountered that are of a previous epoch, preceded by records from the latest epoch. Those records should be skipped during the undo operation. We will see below that they can never be dependent on records of the last epoch.

1.3.4 Client Recovery and Image Snapshots

Traditionally clients replay un-committed transactions, these are communicated to clients by metadata servers trhough last committed numbers. Client recovery interacts with snapshots by starting replay after the rollback has completed.

The MDS nodes collectively determine the last committed snapshot. Clients retain all transactions with transaction numbers that are beyond the last committed snapshot, and free those before. The issue that needs to be addressed is which transactions have been undone and which were not. This information is available from the undo log: each undo operation includes a cancellation of the corresponding undo record.

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.

1.3.5 Avoiding un-necessary 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.

The answer is easy: if the transaction depends on objects from a certain epoch then we need to know that this epoch is globally committed and will not be rolled back during recovery before we can stop writing undo records.

The mechanism for this is to record in the inodes (in a 32bit EA) what the epoch is in which it is being committed when a distributed transaction happens. Transactions that are local to one MDS compare the last globally committed snapshot with the epochs found in the inode numbers. If the epochs in the inode numbers are at least as old as the globally committed snapshot, there is no need to write undo records. The new or modified objects created in this transaction set the dependent epoch EA to the latest epoch found in the dependencies.

1.3.6 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.

1.4 Logic of snapshot coordination

1.4.1 General approach

Our snapshot algorithm is quite cheap, it can be done with 3K messages, where K is the number of targets in the metadata cluster. We propose snapshots are frequently recorded (every second for example) by a rotating coordinating node, called the coordinator.

With the response to a snapshot control message, received possibly asynchronously by the coordinator, the last completed and commited 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 K 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.

1.4.2 Snapshot algorithm

Each undo transaction log will label the undo records as belonging to a **snap epoch, using an integer.** A snapshot is the transition from one epoch to the next. The node will record transactionally which is the current epoch on the node, it will transactionally record when it transitions and finally and at which undo log record number the previous epoch ended, using the mechanisms described above.

Each snapshot needs a coordinator. All K MDS nodes have a server index i and the coordinator for snapshot p is the MDS with index i = p % K.

This node now:

1. Sends a control message to all other MDS servers, this can be done in parallel. The control message handler moves the epoch forward. (message type snapcontrol)
2. Each node reports back to the coordinator when the previous snapshot has comitted (snapstatus message type, with a STATUS_LOCAL flag).
3. The coordinator reports to all nodes when all nodes have committed the previous snapshot (snapstatus message type with a STATUS_GLOBAL and STATUS_PURGE flag). Nodes can now initiate purging of un-needed undo records and stop recording undo information for certain transactions.

These three steps conclude the snapshot.

We have seen above that other nodes may be eager to initiate a snapshot and commit. They can do this with a 4th type of message, sent to the coordinator of the epoch they wish to start. Probably not more than 10 snapshots should be started in a second, and hopefully it is normally a very infrequent operation. Hence:

1. Nodes can initiate a snapshot and send a snapcontrol message to the coordinator.
2. Nodes should not initiate a new snapshot before 100ms have passed since the previous snapshot.

If the snapshot fails to conclude, the coordinator invokes recovery of the metadata cluster.

1.4.3 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 with a snapreqstatus message.

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 snapstatus message with flags STATUS_GLOBAL | STATUS_ROLLBACK to indicate to what point servers should rollback.
5. When this completes messages are received by node i = 1 of type STATUS_LOCAL.
6. When all messages have arrived the coordinator sends a message to all nodes of type STATUS_GLOBAL | STATUS_ROLLB_COMPL.
7. Nodes proceed to accept replay and resent messages.

1.4.4 Snapshot control messages and piggybacked messages

The leader sends a control message to each node in the cluster to start the snapshot. Because commit order and message order is not the same, each node must also piggy back its current epoch number on each request. Compare the Lai-Yang algorithm for meaningful snapshots in message passing systems (cf. Tel, page 343).

The idea of the algorithm is that several actions can trigger the epoch to move forward, and all of them must be honored:

1. A control message initiates a new epoch. All transactions starting after the control message start in the new epoch, as described in detail above.
2. If a request for a dependent operation is received, its receipt treated as a control message followed by the request for the dependent transaction.
3. If a reply to a dependency comes from a node with a higher epoch than the recipients epoch, the recipient
   (a) moves the receiving memory transaction to the higher epoch.
   (b) starts the planned transactions in the new epoch and future planning uses the new epoch.

It is easy to convince oneself that a stack of dependent transactions is pulled into the latest epoch, even if the dependency messages arrive before control messages arrive. This is important, because if the dependent transactions crossed an epoch, undo might remove a partial file system operation.

Also note that not all concurrently running transactions will lie in the same epoch. Once a dependent transaction has reported its epoch to the initiating transaction it must not change its epoch.

1.5 Details of epoch control

Each node manages its epoch number as follows:

1. The epoch increase is atomic wrt:
   (a) starting a memory transaction. But memory transactions can be in progress while the epoch increases.
   (b) sending a message to exchange epoch information and initiating a local transaction and a dependent transaction.
   (c) replies from nodes preparing dependent transactions
2. There are two kinds of increases, the first is called a control increase, the second a dependency increase:
   (a) A control message is sent from the coordinator to another node to move the epoch from n to n+1, if the node has not already done so. We call this a control increase. Nodes that desire to urgently move the epoch forward can do so and request the coordinator to send more messages.
   (b) Each request that
      i. prepares for a dependent transaction, or
      ii. a request which prepares to or reads data for an **fsop** on a dependent node or
      iii. a reply to one of these two

      includes the epoch number of the sender. If the epoch number of the sender is higher than that of the receiver, i.e. if such a message arrives before a control message, the epoch number is increased to n+1. We call this a dependency increase for the receiving transaction and call it a control increase for all other transactions.
3. A running memory transaction is said to adopt the epoch increase if it will label its llog record with n+1, otherwise it is said to ignore the increase.
   (a) On any node, all transactions planned or transactions which were not planned but started after the increase mark the llog records with n+1, ie. they adopt the increase.
   (b) Planned transactions always adopt dependency increases.
4. When the increase is done each node writes an llog to a file which records the epoch boundary. The file contains two integers, the current epoch, and the last written record in the llog file. A reverse scan of the undo log can stop at this record number.

   The epoch boundary record is needed in case a node does not have transactions in the new epoch, and to facilitate an easily determined endpoint of the reverse undo log scan.

Theorem: Assume all nodes in the cluster are in epoch n+1. The effect of all operations caused by llog records with label <= n form an image snapshot.

Proof: If an fsop involves dependent transactions, the algorithm above shows that the entire group of the initiating and dependent transactions fall in the same epoch.

If fsop B reads from A (making B a dependent fsop on A) then the epoch of B is at least as large as that of A, because the reply to the reading operation may cause a dependent increase (see 2-b-(iii) above) .

1.6 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.