Size-On-MDS HLD

Vitaly Fertman

2006-07-21

1 Introduction.

The current HLD introduces the 'Size-on-MDS' metadata improvement, which includes the caching of the inode size, blocks, ctime and mtime on the MDS. Such an attribute caching allows clients to avoid making RPCs to the OSTs to find the attributes encoded in the file objects kept on those OSTs what results in the significantly improved performance of listing directories.

2 Requirements.

2.1 Use case requirements

- ls -l is much faster

2.2 Architectural requirement - IO epochs

An IO epoch mechanism is introduced that allows for two attribute retrieval mechanisms:

- OST based when files are active for IO
- MDS based when files are not active for IO

Broadly speaking an IO epoch for a file starts when a file is opened for writing and closes when (i) the file is closed (ii) the IO caches have been flushed.

Notice that failures in each of these phases can happen due to node crashes (clients, OST's or MDS's).

2.3 Functional requirements

- The MDS caches the file size, blocks, mtime and ctime, it has an authority on these file attributes while files are not opened for write.
- The clients go through the existing mechanisms from LOV to the OSTs to collect the attributes managed by the OSTs on the individual data objects that make up a file, during an IO epoch or if Size-on-MDS caching is not enabled.
- The attribute caching is re-enabled when the IO epoch closes and attributes are updated on the MDS.
- The consistency of the cached attributes and the original values encoded in the file objects is managed in the face of any system crashes.

3 Summary of the solution.

3.1 Current operation.

The MDS has an authority for some inode attributes, but not for all of them. Some of the attributes, such as size, blocks, mtime and ctime, are encoded on the MDS and the file objects kept on OSTs that make a file. Clients to get the file attributes need to make a rpc to the MDS and a rpc to every OST which holds an object of the file.

3.2 Proposed operation.

The MDS has the authoritative file size, blocks, ctime and mtime attribute under normal circumstances, when files are not open on the MDS. Sending them in reply to getattr request from clients, the MDS indicates that attribute caching is enabled. The client, obtaining the cached attributes through getattr requests from the MDS, also gets a read lock over them.

The MDS loses control over the file size when a file is opened for write, and all the given locks are canceled. Since this point the inode revalidation routines will not acquire the proper attribute values from the MDS but instead they have to go through the LOV to the OST's to collect the sizes, blocks and times of the file objects.

The MDS gains the authority back when all the writers close the file and submit the updated attributes on the MDS. All the clients lose the attribute MDS locks again as the result of the attribute update and the following getattr rpc already obtain the cached attributes.

3.3 Enabling the Attribute Caching.

All the operations that may change any of the following attributes – size, blocks, mtime, ctime – on the OSTs need to inform the MDS about the start and completion of the operation. This information is needed to the MDS to disable and to enable caching correspondingly. There are the following operations: write, mmap, truncate.

1. The MDS is informed about the write and mmap at the file open request.
2. The MDS is informed about the truncate at the check permission request.
3. The MDS is informed that the write finished on the close. It also may be able to update attributes if a client sends them along with the close rpc.
4. The client sends attributes with the close only if it has no dirty cache (otherwise blocks is not known yet) and it has attributes under the locks from the MDS and [0;eof] locks from all the OSTs (to not miss utime(2) or a parallel write from another client). If one of these conditions fail, the client does not try to obtain the valid attribute values, it just sends the close rpc without attributes.
5. The client sends WRITE_DONE rpc when its dirty cache is flushed to inform the MDS it is not going to write to objects anymore. The client sends attributes along with the rpc if it knows the valid attribute values. The same rpc is used for the truncate to inform the MDS the operation finished.
6. The MDS updates the inode attributes with the ones sent with close or WRITE_DONE and re-enables attribute caching only if it was an exclusive writer (writes from 2 clients could happen in the order reversed to the order closes come to the MDS and the received attribute values could be already obsolete).
7. When all the clients have flushed their caches and closed the file, the MDS asks the last closer to obtain the valid attribute values and send them back to the MDS with another SETATTR rpc.
8. Having a request from MDS to update the attribute on the MDS, client makes getattr rpc's to OST's which keep the file objects, followed by setattr rpc to MDS.
9. To avoid holding locks over the attributes while a client updating the attributes on the MDS, the MDS contol if any client informs it about coming write or truncate operation while attributes from the OSTs are on the way and just ignore that update if so.

3.4 Logging.

The issue with re-enabling file attribute caching on MDS is one of propagating information about object attribute (file size, blocks, mtime and ctime) changes on OSTs, to the MDS. In order for such information not to be lost somewhere in the middle of the way to the MDS, a llog record needs to be maintained on OSTs:

1. The OST create a llog record transactionally along with the attribute changes on a persistent store(disk).
2. When the MDS commits the attribute update on the persistent store(disk) it also initiates the cancellation requests for the corresponding log records on OSTs.
3. The log record must exists until the cancelation or a destroy request is received on the OST.

3.5 Recovery.

3.5.1 Recovery from a client failure.

The recovery from a client failure is that the MDS performs the client eviction. All the files opened by this client must be closed. If this client was the last writer for a file, MDS must update attributes. As there is no way to get an update from the client, the MDS has to ask the OSTs for the replay logs by itself, and to apply any obtained attribute update to the MDS inodes. After that, the MDS sends out the cancelation requests to the OSTs.

This recovery is transparent for other clients in the cluster, the MDS indicates them it does not have attribute caching enabled on such inodes and the clients obtain them from OSTs.

3.5.2 Recovery from an MDS failure.

At the time of an MDS failover, the MDS does not know which files were opened, which closed but still needed an attrbute update, therefore the MDS attribute caching is disabled for all the files until the synchronisation with the OSTs finishes. The MDS reconstruct the state of inodes during the recovery with clients, they replay to the MDS open, close, WRITE_DONE, SETATTR, etc requests – all that is needed for the inode state reconstruction. The clients are able to proceed their work with the MDS after that.

However, if a client was evicted and the MDS did not finish inode attribute synchronisation with OSTs before the MDS failure, there could be missed attribute updates kept on OSTs yet. Thus the MDS fetches the list of inodes that have

pending attribute update logs on OSTs, obtains the up-to-date attributes for all the inodes that are not opened for write (during the recovery with clients or by client after that), update the attributes and cancel the processed attribute update logs on the OST's.

3.5.3 Recovery from the MDS and an OST failure.

At the MDS failover time, the MDS does not know which files need an attribute update. If the MDS cannot obtain the logs of a failed OST, it does not enable attribute caching for all the files at all. After the OST failover, when the MDS has processed all the OST logs, it enable the global attribute caching again.

3.5.4 The network failure between a client and the MDS.

The MDS performs the client eviction, however if OSTs have not evicted the client, and the client still has a dirty cache, the client may flush it after the MDS recovers all the files opened by the client. To sort this out, the MDS recover such files holding the locks over the file objects on the OSTs.

4 Definitions.

IO epoch. The inode on the MDS is in the IO epoch if somewhere in the cluster an IO can be initiated through interaction with OST's.

IO epoch number. The number that uniquely identifies the epoch an inode is in. The IO epoch number in the inode is initialized by the IO epoch sequencer.

IO epoch sequencer. The MDS global IO epoch number which will be a random number from boot time which is increased each time a new epoch is started on an inode.

Valid Inode Attributes.

- The client has the valid attributes when it has the UPDATE lock on the MDS attributes and the 'Size-on-MDS' caching is enalbed for this inode on the MDS.
- If caching is not enabled, the UPDATE lock from the MDS is not enough, the client must not have a dirty cache and needs all the [0;eof] extent locks from all the OSTs. The lock from the MDS is needed to not miss utime(2), other locks to not miss a parallel write from another client. If a client has a dirty cache, it does not know the correct blocks value yet.

5 Functional Specification.

The global 'Size-on-MDS' flag.

- This is the MDS only flag that indicates if the Size-on-MDS caching is enabled at all.
- It is disabled at the MDS failover time until the MDS gets synchronised with OSTs.

The inode 'Size-on-MDS' flag.

- The flag is set on the inode when its 'Size-on-MDS' caching is enabled on it.
- The MDS sends the flag along with the attributes in reply to getattr requests to indicate the caching is enabled on the inode.
- The clients also set this flag in the inode to remember the MDS has the authority over size/blocks/mtime/ctime attributes.

The inode 'Attribute Change' flag.

- This flag is set into an inode by the client which has changed one of the considered attributes (size, blocks, mtime, ctime).
- The flag is passed to the MDS when the client closes the IO epoch.
- The MDS keeps the flag in the inode to know if an attribute update is needed at the end of the IO epoch.

The 'open IO epoch' flag.

- The flag is sent by a client to the MDS to open an IO epoch before any write IO to the file objects happens.

The 'close IO epoch' flag.

- The flag is sent by a client to the MDS to indicate the client closes the IO epoch. I.e. no IO initiated by this client may happen with the file objects anymore.

The 'OST synchronisation' flag.

- The inode flag that indicates the inode on the MDS is in the OST synchronisation state.

The inode IO epoch holder counter.

- The MDS counts the clients that has opened IO epoch on the inode, those are epoch holders until they close the IO epoch. The epoch is closed on an inode when epoch holder counter is 0.

The inode IO epoch number.

- The inode sets the number of the IO epoch the inode is in when an epoch is opened.
- The inode continues to keep the number of the last IO epoch after the epoch ends to remember what was the last epoch, what is essential in some cases.
- The inode IO epoch number is returned to the epoch opener and is supplied with all the client requests within the epoch, including the epoch close request.

The Attribute Update data.

- The Attribute Update consists of the inode attribute values, the llog record cookies of the OSTs where changes in the file objects took place, and an IO epoch number that the update belongs to.
- The client keeps the data in an inode until send it to the MDS in an IO CLOSE request.
- The MDS keeps it in an inode while they are considered valid until the IO epoch ends and the MDS flushes it on disk.
- The Attribute Update is considered as valid by the client only if the inode attributes are valid.
- The Attribute Update coming on the MDS along with the CLOSE IO epoch request is considered valid only if the 'Attribute Change' flag is not set on the inode yet and no more client sends the 'Attribute Change' flag with the 'Close IO Epoch' request. In other words, an Attribute Update is not valid if at least 2 clients changes attributes within an epoch, because the writes may happen in the order reversed to the order closes come on the MDS and the attributes could be already obsolete.

The 'Attribute Update' inode queue.

- The queue of inodes on the clients and the MDS which the MDS is waiting an Attribute Update for.

The 'Attribute Update' thread.

- The distinct thread on the clients which walks through the 'Attribute Update' inode queue, obtains the Attribute Updates for each inode and sends it to the MDS.

The 'Recovery Attribute Update' inode queue.

- The queue of inodes on the MDS that the MDS needs to obtain Attribute Updates for by itself.

The 'Attribute Update' llog record.

- The llog record on an OST indicates the object attributes have been changed. This is essential part of the 'Size-on-MDS' recoverability.
- The llog record contains the IO epoch number and the object id only.
- OSTs generate a special cookie which uniquely identifies the llog record. It encodes the file object id, which has got an attribute change, and the epoch number. This cookie is used for the quick llog record search the request from the MDS points to.
- The llog record must exists until the MDS has written updated attributes on a persistent store or an unlink occures.
- There is no need to keep a separate llog record for every IO epoch for the same file object. The llog record just maintains the state 'an attribute update is needed on the MDS'.

6 Use Cases.

1. Client obtains the inode attributes. No IO epoch on the inode.
   - The client enqueues GETATTR request to the MDS.
   - The MDS sends the attributes.
2. Client obtains inode attributes. The inode is in an IO epoch.
   - The client enqueues GETATTR request to the MDS.
   - The MDS sends the attributes, but no size, no 'Size-on-MDS' flag, under the UPDATE lock.

- The client performs glimpse requests to OSTs.
- The OSTs send the attributes.

3. Client opens a file for write. No IO epoch on the inode.

   - The client sends OPEN for write to the MDS.
   - The MDS starts the IO epoch.
   - The MDS cancels all the inode UPDATE locks have been given to clients.

4. Client opens a file for write. The inode is in an IO epoch.

   - The client sends OPEN for write to the MDS.
   - The MDS opens the IO epoch.

5. Client truncates a file. The inode is in IO epoch.

   - The client checks the permissions on the MDS.
   - The MDS opens an IO epoch.
   - The client sends punch requests for the file objects to the OSTs.
   - The OSTs truncate objects, create corresponding llog records.
   - The client sends WRITE_DONE request to the MDS.
   - The MDS closes the IO epoch.

6. Client truncates a file. The inode is not in an IO epoch. The Attribute Update is sent along with the WRITE_DONE request.

   - The client checks the permissions on the MDS.
   - The MDS starts an IO epoch.
   - The MDS cancels all the inode UPDATE locks have been given to clients.
   - The client sends punch requests for the file objects to the OSTs.
   - The OSTs truncate objects, create corresponding llog records.
   - The client sends WRITE_DONE request to the MDS.
   - The MDS updates the inode attributes.
   - The MDS re-enables the attribute caching.
   - The MDS ends the IO epoch.
   - The MDS generates the llog record cancelation requests.
   - The MDS sends the cancelation requests out.
   - The OSTs removes the llog records.

7. Client truncates a file. The inode is not in an IO epoch. The Attribute Update is not sent along with the WRITE_DONE request.

    - The client checks the permissions on the MDS.
    - The MDS starts an IO epoch.
    - The MDS cancels all the inode UPDATE locks have been given to clients.
    - The client sends punch requests for the file objects to the OSTs.
    - The OSTs truncate objects, create corresponding llog records.
    - The client sends WRITE_DONE request to the MDS.
    - The MDS returns EAGAIN.
    - The client obtains the inode attributes.
    - The client sends SETATTR to the MDS.
    - The MDS updates the inode attributes.
    - The MDS re-enables the attribute caching.
    - The MDS ends the IO epoch.
    - The MDS generates the llog record cancelation requests.
    - The MDS sends the cancelation requests out.
    - The OSTs removes the llog records.

8. Client closes a file. The last closer. None of clients changed attributes.

    - The client sends close rpc to the MDS.
    - The MDS re-enables the inode attribute caching.
    - The MDS ends the IO epoch.

9. Client closes a file. The exclusive IO holder. The client's cache is flushed. The Attribute Update is sent along with the close rpc.

    - The client sends the close rpc to the MDS.
    - The MDS updates the inode attributes.
    - The MDS re-enables the attribute caching.
    - The MDS ends the IO epoch.
    - The MDS generates the llog record cancelation requests.
    - The MDS sends the cancelation requests out.
    - The OSTs removes the llog records.

10. Client closes a file. The last closer. All the writers flushed their dirty caches.

- The client sends the close rpc to the MDS.
- The MDS ends the IO epoch.
- The MDS returns EAGAIN to the client.
- The client obtains the inode attributes.
- The client sends SETATTR to the MDS.
- The MDS updates the inode attributes.
- The MDS re-enables the attribute caching.
- The MDS generates the llog record cancelation requests.
- The MDS sends the cancelation requests out.
- The OSTs removes the llog records.

11. Client closes a file. Not the last closer. The client has not changed attributes.

12. Client closes a file. Not the last closer. The client has flushed its cache.

13. Client closes a file. The last closer. The client has not changed attributes. Some client still has a dirty cache.

14. Client closes a file. The last closer. The client has flushed its cache. Some client still has a dirty cache.

    - The client sends the close rpc to the MDS.
    - The MDS closes the IO epoch.

15. Client closes a file. The client still has a ditry cache.

    - The client sends close rpc to the MDS.

16. Client flushes its cache. Some client still has dirty cache.

    - The client flushes the dirty cache to OSTs.
    - The OSTs write to objects, create corresponding llog records.
    - The client sends WRITE_DONE rpc to the MDS.
    - The MDS closes the IO epoch.

17. Client flushes its cache. The exclusive writer. Attribute Update is sent along with the WRITE_DONE rpc.

    - The client flushes the dirty cache to OSTs.
    - The OSTs write to objects, create corresponding llog records.
    - The client sends WRITE_DONE rpc to the MDS.
    - The MDS updates the inode attributes.

- The MDS re-enables the attribute caching.
- The MDS ends the IO epoch.
- The MDS generates the llog record cancelation requests.
- The MDS sends the cancelation requests out.
- The OSTs removes the llog records.

18. Client flushes its cache. The last closer. All the writers flushed their dirty caches.

    - The client flushes the dirty cache to OSTs.
    - The OSTs write to objects, create corresponding llog records.
    - The client sends WRITE_DONE rpc to the MDS.
    - The MDS ends the IO epoch.
    - The MDS returns EAGAIN to the client.
    - The client obtains the inode attributes.
    - The client sends SETATTR to the MDS.
    - The MDS updates the inode attributes.
    - The MDS re-enables the attribute cahing.
    - The MDS ends the IO epoch.
    - The MDS generates the llog record cancelation requests.
    - The MDS sends the cancelation requests out.
    - The OSTs removes the llog records.

7 Logic Specification.

1. Inode IO epoch lifecycle.

    - The IO epoch is uniqely identified by a IO epoch number which allows to distinguish one epoch from another.
    - The IO epoch is used only to control the write activity on the inode. The MDS starts a new epoch before the first write IO and ends the epoch when all the clients finished their write operations and flushed their caches, in other words when the IO epoch holder counter becomes 0.

2. Check if caching is enabled.

    - The MDS checks if both global and the inode 'Size-on-MDS' flags are enabled.

3. OPEN IO Epoch.

    - This happens on an OPEN for write request or if an 'Open IO Epoch' flag comes with some request.
    - If IO epoch holder counter is 0, the IO epoch starts.
    - The MDS increments the IO epoch holder counter.

4. The IO Epoch Start.

    - The MDS drops the 'Size-on-MDS' flag in the inode.
    - The MDS gets a new epoch number from the IO epoch sequencer and sets it to the inode.
    - The MDS removes an inode from the 'Attribute Update' and the 'Recovery Attribute Update' queues, the MDS also clears the 'OST synchronisation' flag in the inode.
    - The MDS cancels all the inode UPDATE locks have been given to clients.
    - The client removes the inode from the 'Attribute Update' queue if the inode is in it, if the same client opens the epoch.

5. CLOSE IO Epoch.

    - This happens when a 'Close IO Epoch' flag comes with some request or on a client eviction.
    - The MDS decrements the IO epoch holder counter.
    - The MDS stores in the inode the Attribute Update comes with the CLOSE IO Epoch request if it is considered valid, but not updates the attributes until the epoch ends.
    - The MDS drops the previously considered as valid Attribute Update kept in the inode, if another 'CLOSE IO epoch' request comes with 'Attribute change' flag set.
    - The MDS sets the 'Attribute Change' flag on the inode if receives it in the 'CLOSE IO epoch' request. The MDS always sets this flag in the case of the client eviction.
    - If IO epoch holder counter reaches 0, the IO epoch ends.

6. The IO epoch End.

    - An IO epoch ends when the epoch holder counter becomes 0. This is when the MDS needs to update the attributes and to re-enable caching.
    - The MDS just re-enables the attribute caching if the 'Attribute Change' flag is not set in the inode.

- Otherwise, the MDS updates the inode attributes if the inode keeps the valid Attribute Update.
- If the MDS does not have a valid Attribute Update, it moves the inode into the 'Attribute Update' queue and returns EAGAIN in the reply to the 'Close IO epoch' request – this is the Attribute Update Request.
- The response to an Attribute Update Request is a SETATTR request that contains an Attribute Update of the IO epoch the client has closed. If a SETATTR request related to another IO epoch comes, the MDS skips it. This avoids holding OST attribute locks while performing an attribute update and saves the MDS from obsolete Attribute Updates from previous epochs if the case abother epoch starts while waiting for the Attribute Update.
- The MDS updates the attributes when an expected Attribute Update comes and it is considered valid, i.e. there is at least one cookie in it. Otherwise, there is no need to perform an update.
- The MDS re-enables the attribute caching, i.e. sets 'Size-on-MDS', clears 'Attribute Change' flags in the inode and removes the inode from the 'Attribute Update' queue.
- The MDS updates the attributes on the persistent store and after that generates the cancelation llog record requests for all the cookies received in the Attribute Update. The MDS packs the proper cookie into every request.
- It also drops the Attribute Update kept in the inode.
- The MDS sends the cancelation requests out to the OSTs.

7. The 'Attribute Update' llog record lifecycle.

   - The llog record is created transactionally along with the attribute change on the persistent store on the first write IO on the file object.
   - The following write IOs do not change the llog record until one of the next epoch come, in which case the OST schedules a transactional llog record epoch number update along with the object attribute change.
   - The log record exists until either a cancelation request from the MDS comes, indicating the MDS has committed the Attribute Update to the persistent store, or a destroy request comes.

8. GETATTR.

   - The client sends GETATTR request to the MDS.
   - The MDS checks if the 'Size-on-MDS' flag is set. If not, the MDS checks if the inode is in an IO epoch or is in 'Attribute Update' or 'Recovery Attribute Update' queue. If not, the MDS sets the 'Size-on-MDS' flag on the inode.

- The MDS packs the attribute into the reply. It packs size, blocks and the 'Size-on-MDS' flag in the reply only if the attribute caching is enabled on the inode.
- The client sends GETATTR request to the OSTs, if it does not obtain 'Size-on-MDS' flag from the MDS. This returns not attributes only, but the complete Attribute Update, i.e. the OSTs pack into the replies the cookies for the llog records if they exist. This feature is used in the Attribute Update Request and in the MDS synchronisation after failover.
- The client merges the Attribute Updates obtained from the OSTs and keeps the result in the inode, only if the client is an IO epoch holder. Otherwise, only attributes are kept.
- The client sets 'Size-on-MDS' flag to the inode if obtains it from the MDS, otherwise it clears it.

9. OPEN for write.

   - The client sends the OPEN for write request.
   - The MDS opens the IO epoch and sends the IO epoch number in the reply.

10. TRUNCATE.

    - The client sends SETATTR request to the MDS to checks the permissions and updates the mtime/ctime. The client sends 'Open IO epoch' flag along with this request.
    - The MDS opens an IO epoch and sends the IO epoch number in the reply.
    - The client sends punch requests for the file objects to the OSTs along with the IO epoch number.
    - The OSTs send the Attribute Updates in the replies.
    - The client merges the Attribute Updates from all the OSTs and keeps the result in the inode.
    - The client sets the 'Attribute change' flag in the inode.
    - The client prepares and sends the Close IO Epoch Request and sends it along with the WRITE_DONE request to the MDS.

11. CLOSE.

    - The following is performed only if the inode is in an IO epoch, i.e. opened for write.
    - If the client has no dirty cache, it prepares the Close IO Epoch Request, it will be sent along with the CLOSE request.

12. FLUSH.

    - The client flushes its dirty cache to OSTs, sending along the IO epoch number.
    - The OSTs send the Attribute Updates in the replies.
    - The client merges the Attribute Updates from all the OSTs and keeps the result in the inode.
    - The client sets the 'Attribute change' flag is set in the inode.
    - The client prepares the Close IO Epoch Request and sends it along with the WRITE_DONE request.

13. Prepare the Close IO Epoch Request.

    - The client packs the 'Close IO epoch' flags and the IO epoch number to the request.
    - The client also packs the 'Attribute Change' flag, if it is set in the inode.
    - The client also packs the Attribute Update kept in the inode if it is considered valid.
    - The client does not try to obtain the valid attributes if it does not have them.
    - The client drops the Attribute Update kept in the inode and clears the 'Attribute Change' flag.

14. Attribute Update Request.

    - The client may receive EAGAIN error on the IO epoch close request. This indicates the MDS asks it for the Attribute Update for this epoch.
    - The client does not remove the relevant requests from the request queue (OPEN and CLOSE or WRITE_DONE) – it allows the client to re-send these requests to the MDS during the MDS recovery and to reconstruct on the MDS the state the inode is waiting for a reply to the Attribute Update Request. They will be removed later when a confirmation the Attribute Update is flushed on disk on the MDS is received.
    - The client puts the inode into the 'Attribute Update' queue.
    - A distinct 'Attribute Update' thread on the client walks through the queue and performs the attribute update for every inode in it.
    - The attribute update involves the GETATTR requests to the OSTs that keep a file object to obtain their Attribute Updates, the client merges them and keeps the result Attribute Update in the inode.

- If ENOENT is returned from an OST, the inode is being unlinked and there is no need for an Attribute Update on the MDS anymore, the client removes the inode from the 'Attribute Update' queue, drops the Attribute Update kept in the inode and clears all the relevant flags in the inode.
- If the inode is still in the 'Attribute Update' queue, the client sends the Attribute Update kept in the inode in an SETATTR request. The request is sent with the IO epoch number that the client closed before.
- The client drops the Attribute Update from the inode and removes it from the Attribute Update queue.
- To perform the update, the MDS will need to merge these OST attributes with its own attributes. No lock is kept over the attributes, if another IO epoch starts, the MDS will skip this update.

15. UTIME.

    - Even if the client holds the IO epoch opened, the client sends a simple SETATTR request, only new set attributes with no cookies, etc. In this case, there is no special handling of utime(2).
    - If the inode is waiting for an Attribute Update on the MDS, it is update of the OST attributes whereas utime(2) changes MDS attributes only. Thus the coming attribute update can be easily applied after that.
    - As the attribute update is applied to the inode at once, there is no need to remember the attributes are changed, moreover it does not concern the OST attributes.

16. UNLINK.

    - The MDS removes the inode from the 'Attribute Update' and 'Recovery Attribute Update' queues and clears all the relevant flags on it. It also drops the Attribute Update kept in the inode.
    - The MDS removes all the cancelation requests not yet sent to the OSTs.
    - The OST removes the 'Attribute Update' llog record when a destroy request comes for the corresponding file object.
    - The client removes the inode from 'Attribute update' queue.

17. Client eviction.

    - The MDS closes IO epochs on all the files opened for write by the client. The MDS sets 'Attribute change' flag on them to not miss a change made by this client.

- The IO epoch may end for some of these files. The MDS moves such inodes into the 'Recovery Attribute Update' queue.
- The MDS walks through the 'Attribute Update' queue and moves all the inodes, that wait for an Attribute Update from this client, to the 'Recovery Attribute Update' queue.
- The MDS drops the Attribute Updates from those inodes that are moved to the 'Recovery Attribute Update' queue.

18. The MDS failover and synchronisation.

    - The MDS drops the global 'Size-on-MDS' flag until gets synchronised with all the OSTs.
    - After the clients re-connect, they replay the pending requests to the MDS. All the requests that open or change a file within an IO epoch, contain the IO epoch numbers. The MDS reconstructs which files are opened, which are waiting for Attribute Updates, setting the correct IO epoch number, IO epoch holder counter, dropping the 'Size-on-MDS' flag on the inodes, populating the 'Attribute Update' queue, etc.
    - The MDS sets the 'Attribute Change' flag to all the opened files. The MDS does not know reliably what clients were connected to the MDS before the failure, thus this is impossible to detect if a failed client changed and closed the file before the failure. Only after that the recovery with the clients is finished. The failed client is evicted from the cluster at the end of the MDS failover.
    - The MDS initializes the epoch sequencer to the last IO epoch number occurred in the client replays.
    - The MDS obtains from the OSTs the list of inodes that need an Attribute Update. The MDS also obtains the IO epochs numbers along with the inode numbers. All the needed info is kept in the llog records on the OSTs. The MDS indeed needs a synchronisation with the OSTs again due to not 100% reliable information what clients were connected to the MDS before the failure.
    - The MDS considers an inode needs a failover Attribute Update if the inode is not in an IO epoch and if not in the 'Attribute Update' nor 'Recovery Attribute Update' queue yet, if all the IO epoch numbers obtained from the OSTs are not greater than the number that epoch sequencer was initialised to at the failover time.
    - The MDS moves the inodes that needs a failover attribute update into the special 'Recovery Attribute Update' queue, setting the 'OST synchronisation' flag to them.

    Another reason for the MDS to synchronise with OSTs is Attribute Update kept in inodes not from the last IO epoch closers. If this is the only

attribute change occurred within the epoch, the MDS flushes this update to disk. However, the request from the first client does not lead to the transaction, from others do not contain the Attribute Update at all – these requests are not kept in the replay queues on clients. Thus if an MDS failure occures, the MDS needs to check there is no missed pending Attribute Updates on the OSTs.

19. 'Recovery Attribute Update' queue handling.

    (a) Client eviction.

        - The MDS makes ENQUEUE GETATTR requests for every inode in the 'Recovery Attribute Update' queue, obtaining the Attribute Updates for them.
        - It is not a simple GETATTR here because if the problem is in the connection between the client and the MDS, the eviction does not happen on OSTs and the dirty cache the client may get eventually flushed to the OSTs. To not be left with a wrong attributes on the MDS, the ENQUEUE request forces clients to flush their dirty caches before obtaining the attributes.
        - The MDS always perform the attribute update with values obtained with the ENQUEUE GETATTR in the 'Recovery Attribute Update' queue even if the inode IO epoch has changed. It is possible if another client opens the file, writes to the file objects and fails – then inode may get into the 'Recovery Attribute Update' queue again. However, the OST attributes are obtained under locks so values are correct.
        - The MDS skips the attribute update obtained with the ENQUEUE GETATTR if the inode is already not in the 'Recovery Attribute Update' queue at the moment of receiving reply.
        - The following steps are similar to the ones in 'the IO epoch end', except the queue is 'Recovery Attribute Update' one.

    (b) The MDS synchronisation.

        - The MDS makes GETATTR requests for every inode in the 'Recovery Attribute Update' queue that has a 'OST synchronisation' flag set, obtaining the Attribute Updates.
        - The MDS updates the inode with the obtained Attribute Update only if the inode still has 'OST synchronisation' flag set. Otherwise, an IO epoch has been opened on this inode.
        - The following steps are similar to the ones in 'the IO epoch end', except the queue is 'Recovery Attribute Update' one and the MDS clears the inode 'OST synchronisation' flag too.
        - The MDS sets the global 'Size-on-MDS' flag, when the 'Recovery Attribute Update' queue has no more inodes with the 'OST synchronisation' flag set.

20. The MDS and a client failover.

    - The actions are similar to the ones in 'the MDS failover and synchronisation'. Only the difference is mentioned.
    - The MDS sets the 'Attribute Change' flag to all the opened files. This is needed as it is impossible to detect if the failed client changed and closed the file before the failure. Only after that the recovery with the clients is finished. The failed client is evicted from the cluster at the end of the MDS failover.

    Question: the IO epoch sequenser cannot be correctly initialized if a client evicted. The recovery with the clients cannot be finished until the MDS gets synchronised with all the OSTs.

21. The MDS and an OST failover.

    - The MDS performs the failover. However, it does not know if some file still needs an Attribute Update until all the OST have sent their llog records to the MDS.
    - The MDS re-enables the global 'Size-on-MDS' flag, when it obtains the 'Attribute Update' llog records from all the OSTs.

22. The MDS, a client and an OST failover.

    - The client will be evicted from the cluster at the end of the MDS failover.
    - The global Size-on-MDS caching cannot be enabled until the MDS gets synchronised with all the OSTs.

    Question: the MDS cannot finish the recovery with the clients until gets synchronised with all the OSTs, but at the same time it cannot synchronised with all the OSTs. The system does not start.

8 State Specification.

probably: inode on MDS (quiescent, opened for write, IO epoch, waiting for Attribute Update).

probably: 'Size-on-MDS' flag on an inode on the clients (attributes are cached, canceled on open for write, attributes not cached, cancelled on attribute update/left not cached if an IO epoch ends with no attribute change).

9 Alternatives.

1. Make CLOSE and WRITE_DONE rpc asynchronous.

   For the better performance it is possible to make close and write_done rpc asynchronous.

2. Atime on MDS.

   The attribute caching on the MDS may also include the atime caching. However, in contrast to the attributes discussed above, the atime is changed on read, so the MDS needs to control open for read requests. The atime is not so important and it is acceptable to lose its change – no OST logs are needed. All the readers are able to propagate the atime changes to the MDS on the close requests, and the MDS chooses the largest one. When all the readers close the file, the atime flushes on the disk.

   The atime update from the readers may conflict with the explicit setattr call when set to the past (utime(2)). However, this lays out of the scope of the 'Size-on-MDS' issue and rather belongs to the bug #10641 problem.

   Adding atime to the attribute caching on the MDS, the caching cannot be enabled when a file is opened for read as well as for write, because even in the case of open for read, the client needs to make extra rpc requests to OSTs anyway.

3. Getting the attribute update on the MDS.

   The file is made up of objects stored on distinct OSTs. To update inode attributes, the MDS needs to know the attribute update made by clients and the cookies from all the OSTs whose objects have been changed. As was showed above, the MDS cannot trust the attribute update of the last closer if it was not an exclusive writer. However, an optimisation can be made here.

   - There is no need in extra Attribute Update Request in the following cases.
   - If times only are changed, the MDS can easily chose the larger one.
   - If size or blocks are the same in all the Attribute Updates, there is no conflicts.
   - Even if size or blocks are different but ctime is also different, the larger ctime points to the later update.

   Thus, only in the case of different size/bocks when ctime is the same, an extra attribute update is needed.

4. Attribute update from a previous epoch.

   What the MDS should do when getting an attribute update of a previous epoch when another epoch is active on the inode? Having in mind that

the current epoch may finish without any attribute change, it is porbably possible to avoid an attribute update cycle at the end of the current epoch by storing the received attributes on disk.

5. Attribute Update Requests on the MDS failover.

   There are several solutions for the MDS to obtain the attribute update for the inodes which are waiting for replies to Attribute Update Requests:

   - The client keeps OPEN and CLOSE requests to the MDS during the recovery and allows the MDS to reconstruct the inode state (the current way).
   - The MDS finishes the recovery after all the clients send their attribute updates. Although this depends on the clients and OSTs interactions.
   - The clients kill their 'Attribute Update' Queues on the MDS recovery.
   - The MDS and clients do not do any special actions, this means the clients send the attribute updates after the MDS recovery finishes, whereas the MDS obtains the attribute update from the OSTs.

6. The OST Synchronisation.

   It happens sometime that OST is down for the long time awaiting a hardware replacement or a such. To minimize the preformance degradation for the case such an OST needs a synchronisation on the MDS, it is possible to disable the 'Size-on-MDS' flag in the inode only if one of its OSTs is down, instead of disabling it globally on the MDS.

10 Focus on Inspections.