VERSION-BASED RECOVERY HLD

PETER BRAAM, MIKE PERSHIN

1. REQUIREMENTS

(1) A server only replays a client update request when the update affects exactly the same versions of objects as it did in the original execution.
(2) A client is guaranteed that all information that it has obtained from the server is present after recovery.

2. FUNCTIONAL SPECIFICATION

2.1. **Current operation - all client, all transaction recovery.** Currently Lustre meets requirements 1 & 2 by requiring all clients to replay all available transactions in exactly the same order as the original execution before any other requests are executed.

Additionally the server only allows clients to recover during a specified period. If any one client doesn't connect to the server during the period,

(1) recovery will abort and
(2) all clients will be evicted at recovery time and
(3) any client connecting after the recovery window closes will be evicted when it tries to reconnect.

In some cases, e.g. Catamount or after long disconnections, clients can't send requests to server in time, in such cases eviction is guaranteed.

2.2. **Version based recovery.** The two requirements will be met if

(1) During replay of requests version-based recovery will allow clients to replay if and only if the objects the client is using have exactly the same version as during the original execution.
(2) After replay, recovery can complete successfully if the data *clients* obtained from servers before recovery is assured to be present and not to have rolled back. A key issue, subject to policy is if, *client* designates the *client Lustre file system* or *client applications* using the file system.

If both conditions can be met jobs can continue without errors. If the conditions cannot be met, eviction will be the normal result, but more relaxed recovery options can be made available to the client.

2.2.1. *Definitions:*

epoch:: boot cycle sequence number
version:: `{epoch, transno}` pair labeling objects (MDT inodes, OST objects). Version is changed to mark the modification of object. Not any modification produces version change, e.g. changing of time or size.
preop-version:: the version of the inodes when it has just been locked before the operation.
postop-version:: the version of the inodes as set by the operation.
version-based recovery:: replays will be done with version checking.
primary recovery stage:: first recovery stage, when server waits for all clients to reconnect and tries to replay them in transaction order (used now as only recovery way)
delayed recovery stage:: recovery of clients which weren't able to reconnect in time and doing that later.

2.2.2. *Summary of version-based replay.*

(1) The server records the last transaction number it has executed for each client transactionally on disk in last_rcvd. A client reconnecting at any time can be told what requests it should replay and which it should discard.
(2) the client allocates an extra space in the reply buffer of the ptlrpc_request ('req->rq_repmsg') to store the versions that are related to the request.
   (a) note: such information is also useful for *undo operations.*
(3) The server collects the preop versions during request processing and sends these versions along with normal reply so that the client can send these versions to the server during replay.
   (a) NOTE: Due to the frequent use of LOOKUP IBIT locks, which do not protect the version of an inode, the client normally doesn't know the pre-op versions when originally executing an operation.
(4) when the client connects to the server for recovery, the server will recognize is it the primary recovery phase or the delayed recovery is needed and what the client last committed number is.
(5) during replay client sends the requests with a transno after the last committed transno to the server.
(6) during replay client will send the pre and post operation versions to the server along with normal request message. Pre-version is used for version checking and postop version will be stored as new one.
(7) during replay the server will attempt to execute every request the client offers, even after it encounters a reintegration failure.
(8) during replay the server checks the objects preop version before replaying a request: server compares the preop version of the inode with the preop versions in replay request. If the version is compatible, then the replay request will continue; if the versions do not match the request replay will get -EOVERFLOW
   Note:
   (a) Even after getting this error on a reintegration, the server will process subsequent requests in the same manner.
   (b) While no gap has been encountered it is not necessary to check versions. Though it is not harmful and Lustre can check versions always but it will fail if old client is connected which sends no versions.
   (c) If a process has an open file handle or active file system lock and versions for the inodes do not match, then the client can optionally kill the process if it receives an -EOVERFLOW message.
(9) When finished with replay, client and server check if any replay failed on any request because of version mismatch. If not, the client will get a successful reintegration message. If a version mismatch was encountered, the client must be evicted.

2.2.3. *Last_committed managing.* The version recovery differs from normal one in transaction management. The problem is that server's transactions are in new epoch already but old client replays are with old epoch's transactions. The problem is how to determine which replays are committed already. E.g. client will start recovery and will use old transno, but first server last_committed reported to the client will be bigger than any replays transno and all replays will be dropped.

The solution is the per-export last_committed value rather than per-obd one.

(1) The commit callback updates export last_committed value in the same way as it does for obd last_committed
(2) the server send the last-committed from export in reply
(3) The aborted recovery will restarts correctly because client will get own last_committed value.

2.2.4. *Last dependency checking.* The recovery transaction gap problem is that if the replay sequence is not complete it is possible that a client has used information that was generated during the gap and the client can no longer get access to this information after recovery.

There are two approaches to complete correctness to this issue:

all transactions, all clients: If it is known that all clients replayed all transactions in the same order, any gaps present in the sequence are harmless (they arise from lost replies).

newest client_object before gap: If a client can report it only had used server data older than the last transaction before the first gap in the sequence, then the client cannot have seen data that was lost. This is best verified by the client remembering, **not** the last transaction number, but the version of newest object it ever obtained from the server - if that object is from before the first gap, the client is not depending on any information lost in a gap.

Another approach can guarantee that the client file system data in the client kernel does not contain any information that was generated and lost corresponding to a certain gap.

cached objects newer than gap correct: The client verifies that all objects in its cache newer than the first gap have exactly the same version on the client and the server

The problem with this approach is that, while the file system data on the client is consistent with what the server has, cache entries may have been present on the client in the past. If these were used and eliminated prior to checking this version, the client applications have used data that was lost on the server. Such applications may even have exited altogether already. Such cache purges can happen before the server failed and also while the client was waiting to reconnect cached data that was used could be purged from a cache.

A weaker check can still guarantee a file system that is consistent between a client and a server - the client purges all unused cache entries and then performs:

Used cache objects newer than gap correct: All objects in the cache that are in use have the correct version.

Note that killing processes can assist with this condition to be satisfied.

The following summarizes the algorithm:

(1) The server maintains a persistent record of the last transaction before the first gap during primary recovery. Notice that the gap can shrink during delayed recovery. Such updates will be recorded transactionally.
(2) When all clients are connected in normal mode or when an operator chooses, the earliest gap information can be cleared.
(3) Every client maintains the last transaction that the server reported as executed during normal processing and the version of the newest object the client has ever obtained from the server.
(4) When the replay has completed, the server and client compare the transaction on the first gap and the newest version the client has seen. If the newest version is earlier than the first gap, the client is up to date. This leads to *full node recovery*.
(5) **If the gap closed during delayed recovery but not all clients have reconnected, further delayed recovery uses version checking.** The client then compares each of its cached objects newer than the first gap with server versions. If the servers are equal or newer the client can completely recovery. This is called *full file system recovery.*
(6) Finally the client can purge its cache and repeat the previous step for the cached objects that are in use. If this test succeed the client performs *weak file system recovery.*
(7) **Optionally** the client can request eviction if full node recovery or full file system recovery cannot succeed.
(8) **Optionally -** If the client can record in each process what the newest object is that the process may have depended on in a system call, then the client can determine which processes have seen stale data, although processes may have exited already and seen stale data. Optionally all processes that have seen stale data could be killed.

2.2.5. *Summary of reintegration possibilities.* It seems that reintegration can go through increasingly weaker steps, all of which are important and interesting. Only the first 2 are required at this stage, 3-5 will be mandatory when writeback caching and disconnected operation is introduced later.

(1) when all clients are present we have guaranteed correct reintegration from the application perspective. (current functionality)
(2) when there is delayed recovery with full reintegration and the last dependent transaction falls before the first gap we get the same.
(3) when there is delayed recovery with full reintegration and versions are correct we have full file system recovery
(4) if there is no export we have no last_rcvd information from the client, so the client doesn't know where to start replay. We can try to reintegrate but likely many re-integrations will fail on version mismatch. However, if at any point during the reintegration we see that the client request pre-op version or the client reply post-op version equals that of the server, that means that for that inode reintegration can succeed. Likely all the transactions we tried to reintegrate before this condition held were committed on the server, the ones after were not. To do this one would need to retain a list of inodes on the client that have seen conflicts during replay; items on the list can be cleared by the client from the list if one of the conditions was met. This can be followed by a version check as above. While theoretically interesting, it is very cheap for the server to retain a persistent client export, even for millions of clients), so retrying transactions without the last received knowledge is probably not so valuable.
(5) If clients get persistent caches the validation routine will become very expensive - possibly millions of inodes need a version check. This can be made efficient: the sub-tree change time aids with traversal of a tree. The client descends through the cache and only when the cache change time is different from the server does the sub-tree need further validation. These subtree change times act as persistent memories for a writeback lock: if they didn't change, nobody used a write lock in a sub-tree to change an object in the sub-tree. But they are fairly expensive to update - every inode in the path to a modified inode would get a subdir ctime change.
(6) Optimistic reintegration of clients facing conflicts is also very interesting. Operations can continue as long as no semantic conflicts are found. For example a file creation can be replayed if the name was not created by another client, _and_ if the permissions on the ancestor directories were no changed. The Coda literature discusses a full set of conflict conditions for

this case (but the AFS/Coda authorization mechanism is through an ACL just on the parent, not on the ancestors to the root of the FS). User interaction is probably needed to clear the conflicts.

2.2.6. *Inode version data.* This is *server maintained* pair {epoch, transaction}. Epoch is boot sequence number. Transaction is the last one which the object was involved in. The comparison of the versions of two different inodes is meaningful. Higher {epoch, transno} means latest version of the object on current server.

2.2.7. *Setting the new version.* The new version is set when transaction is assigned for the operation and is written into last_rcvd file. Right before setting new version the old one should be saved as preop version in reply. This is important for parent directory mostly because it can be used by several threads at once due to pdirops feature, therefore the only way to get correct preop version is reading it right before writting the new one because that is serialized.

2.2.8. *Request structures.* All request structures will have room for pre-op and post-op versions of 1-4 inodes (4 are required for a rename that is clobbering another inode). Metadata requests will be structured in such a way that they contain full re-do information including version checking, and have all information required for undo.

2.2.9. *Reply structures.* All replies for requests will include the pre- and post-op versions.

2.2.10. *Resend.* Last_rcvd should keep preop version. Reconstruct resend will use it to restore reply properly

2.2.11. *Versions after reconnect.* After rebooting the server starts new epoch with transactions started from 1. This way allows to don't know the last used transaction before recovery and server doesn't depend on disconnected clients.

2.2.12. *Assigning version upon delayed replay.* we should still have separate transaction field in reply among with pre-op version and post-op version. Post-op sets new version and transaction is transaction of operation. In case of delayed recovery it will be new transaction in new epoch. Therefore we will be able to check committed request on client, and able to replay it again if needed in new epoch, but versions are preserved. So transno is used to track request state(committed or no) and for recovery, but version is updated from postop version.

2.2.13. *ptlrpc_check_dependency(struct lustre_import *).* This function returns -EOVERFLOW unless one of the following is true:

- the server has replayed all transactions before the last transaction the client depended on OR
- all clients were present during recovery

When this function is called the state of the recovery can change to *dependency_mismatch*

2.2.14. *ptlrpc_recovery_success(struct lustre_import *).* This function returns success unless the state of the import or export is:

- dependency_mismatch OR
- reintegration_failed

3. State Management

3.1. **Epoch management.** The epoch controls what clients should participate the main recovery. The following rules are used for epoch management:

- Epoch is increased upon every server boot cycle when recovery ends.
- Any client with epoches older than last one are not included in recovery and marked as delayed.
- The every connected client writes current epoch in client data in last_rcvd file when completes recovery.

3.2. **Main recovery changes.** Main recovery phase is not ended if some client(s) miss recovery window. Instead of eviction the other clients continue to recover with version-based recovery.

- the recovery is timed out
- for missed client as exp_delayed = 1;
- continue recovery with VBR (obd->obd_version_recov = 1)
- every client with version mismatch is evicted when finishes recovery

3.2.1. *Failure during main recovery (no VBR).* The server didn't update the last epoch yet, so recovery will restart with the same conditions.

3.2.2. *Failure during main recovery (VBR).* VBR was turned on so there was gap, i.e. one (or more) client didn't connect in time. After failure there are two possible cases:

(1) Missed client will not participate in recovery again. The gap will be determined again and recovery proceed as before failure.
(2) Missed client will connect to server:
    (a) client will participate in recovery if its epoch is the same as server last_epoch or wait for main recovery to complete if otherwise
    (b) The per-export last_committed will be used to control client replays are needed or committed already.

3.3. **Delayed recovery.**

(1) Upon connection the last_transno for this client is got from last_rcvd file
(2) The last_committed transno is sent to the client.
(3) Client gets its last_committed value and drops old replays and sends others to server.
(4) Server processed replays with version checking updating per-export last_committed value.
(5) If version mismatches are seen then client is evicted. Reintegration fails.
(6) Locks are not replayed during delayed recovery as they can be already obsoleted.
(7) If recovery was successfull then client's epoch is set to current one. Client is now fully reintegrated.
(8) All normal requests are blocked from execution during delayed recovery to avoid unnessessary conflicts:
    (a) to avoid conflicts in case of recovery failure. E.g. normal request uses the same object as was modified during delayed replay. In case of failure the main recovery replays will depends on delayed client's replay. Note: the COS will solve this as dependent replays will be synchronous;
    (b) to increase the possibility of successfull reintegration because normal requests may change version of the same object as delayed client is going to change during replay.

3.3.1. *Failure during delayed recovery.*

(1) The server is rebooted and read client's data from last_rcvd.
(2) The delayed client's epoch is not yet updated because it didn't finish recovery, so it is excluded from main recovery window.
    (a) The replays from this client which were done before failure don't affect the replays of other clients because they passed version-checking and all normal replays were prohibited during delayed recovery.
(3) When main recovery is finished the delayed client is allowed to connect and delayed recovery starts again.
(4) As client has own last_committed value it proceed in the same way as before the failure.

3.4. **Import recovery changes.** Import control the return code from server to see -EOVERFLOW. If it happens then imp_vbr_failed is set to show that reintegration fails. The recovery process will continue to replay all requests but locks will not be replayed and client will be evicted after all.

3.5. **Disk format changes.** Inode will store version on disk, it is scope of another HLD

3.5.1. *Last_rcvd file changes.* client_data contains extra data now:

(1) last_epoch this client was seen
(2) pre-versions of last operation

3.6. **Wire format changes.** Request and reply structures are changed and contain version fields for all objects involved in operation

4. USE CASES

4.1. **primary recovery steps.**

(1) server completes its initialization and starts to accept connecting request from clients. The server, as before goes into recovery mode if old exports are found.
(2) server waits TIMEOUT+X seconds to allow clients to connect, those clients which connected with server are deemed to be normal recovery clients. Each time a client connects the recovery connect window is grown up to a maximum value.
(3) In the connection handshake the server reports to the client what the last transaction is that it has committed.
(4) The server begins to receive replay requests from clients, if the transno of the request is in the right order it continues to process it.
(5) After a gap is encountered, the server does not proceed until at least TIMEOUT+X has elapsed since the last reconnect, to allow other clients to join in and close the gap.
(6) After a gap wait replay continues and version checking for integration will be used and block or allow the request to be processed based on its version.
(7) when a client completes its replay successfully it performs a dependency check with the server and completes recovery or evicts itself
(8) when server completes recovery all obd_exports for still disconnected clients are retained - they are needed to find the last_transaction executed for a client during later replay.

4.2. **obd_export maintenance.** We need to maintain last_committed for not-connected clients so they can understand what was committed already. That is why the info about non-connecting client should be retained.

(1) Exports for disconnected clients can be cleaned up by an lctl command
(2) If a client at a nid connects with a new UUID any old export can be cleaned up
(3) Perhaps no more than a certain amount of disconnected exports are retained

4.3. **Gap recording.**

(1) When encountering a gap, when not all clients are present the server will record the gap data with transaction that is following the gap.
(2) If all clients are present during recovery and replay completes the gap information, if present, is cleared
(3) An lctl command is available to clear gap information

4.4. **Issue with permissions.** As it's possible to lose some changes security issues arise. For example, user can close access to subtree /A/B and then create secret file /A/B/C/D/file. If previous version of /A/B is lost, then user's setattr changing permission can't apply while nothing prevent secret file creation from replay. Thus secret file may be exposed after recovery. As there is no cheap way to track dependency in this case we propose two options: make all requests changing permission synchronous and allow user to make them asynchronous. Actual behavior is controlled via procfs-exported *sync-permission* variable.

5. LOGIC SPECIFICATION

5.1. **Inode version life-cycle.**

- the last transaction of inode and server epoch are used to maintain versions, the modified ext3 is used to store version in inode.
- add/del entry in directory will update the version of directory inode
- data modification will update the object version with new transno and this must match for delayed writes of dirty data to be allowed; that decision is made at lock replay time.
- when file size, mtime or atime transfers from the OST to the MDT the MDT inode version will NOT be updated.

5.2. **SETATTR.**

5.2.1. *normal mode.*

- record preop-version into ptlrpc_reply after the inode is locked
- check the COS is needed and do synced operation if so
- setattr on the inode, update inode's version
- record postop-version into ptlrpc_reply before the inode is unlocked
- if setattr changes permission bits or uid/gid and *sync-permission* variable is set, then mark transaction synchronous and wait for commit on the server side

5.2.2. *replay mode.*

- inode's version is equal to preop-version, then continue to process
- else return -EOVERFLOW

5.3. **CREATE.**

5.3.1. *normal mode.*

- initialize the version of the new created inode after it is locked
- get preop-version of the parent inode
- update the version of the parent inode
- record updated version of parent inode before it is unlocked

5.3.2. *replay mode.*

- lock the parent inode
- if parent inode's version matches the preop-version in the replay request then continue to create and update the child's version
- else return -EOVERFLOW

5.4. **LINK.**

5.4.1. *normal mode.*

- record the preop-version of the source inode and the target directory inode after it's locked
- update the version of source inode and the target directory inode
- record postop-version of source inode and target directory inode

5.4.2. *replay mode.*

- if the version of the source inode and the target directory inode match preop-version in the replay request, then continue to process
- else return -EOVERFLOW

5.5. **UNLINK.**

5.5.1. *normal mode.*

- record the preop-version of the parent directory inode and the child inode
- check the COS is needed and do synced operation if so
- unlink the child inode and update version of the parent&child inode
- record the postop-version of the parent directory inode and the child inode

5.5.2. *replay mode.*

- if the preop-version between parent&child inode and replay request match, then continue to process
- else return -EOVERFLOW

5.6. **RENAME.**

5.6.1. *normal mode.*

- record the preop-version of the two parent inodes, the source inode and the target inode if it exist after it are locked
- check the COS is needed and do synced operation if so
- update the version of related parent&child inodes after operation
- record the postop-version before it are unlocked.

5.6.2. *replay mode.*

- if the preop-versions match, then continue to process
- else return -EOVERFLOW

5.7. **OPEN.**

5.7.1. *normal mode.*

- record the preop-version of the directory inode after it is locked
- check the COS is needed and do synced operation if so
- if the child inode exists, then record its preop-version
- if the child inode doesn't exist, then initialize version of the new created inode
- if the child is created, then update the version of the directory inode

5.7.2. *replay mode.*

- if preop-version match, then continue to process
- else ereturn -EOVERFLOW

5.8. **ENQUEUE.**

5.8.1. *normal mode.*

- record the preop-version of related inode after it is locked

5.8.2. *replay mode.*

- if preop-version match, then continue to process
- else return -EOVERFLOW

5.9. **OST_DESTROY.**

5.9.1. *normal mode.*

- store preop-version of the obdo after it is locked

5.9.2. *replay mode.*

- if preop-version match, then continue to process
- else return -EOVERFLOW

6. Alternatives

7. Focus of Inspection