High Level Design

Mike Pershin, Alexander “Zam” Zarochentsev

August 13, 2008

1 Introduction

1.1 Definitions

transaction an atomic operation which reads and possibly writes lustre filesystem state including file and directory attributes, directory entries and file contents

uncommitted data filesystem state updated by one or more transactions but not yet committed to stable storage

inter-client dependent transactions (also **dependent transactions**) transaction (B) depends on transaction (A) if:

1. (B) reads from an object after (A) has written to it.
2. (A) and (B) are issued by different clients

recovery connection re-establishment by a client after a server failure. Recovery is successful if all of the client’s uncommitted transactions can be re-executed correctly and its cache can be revalidated. If recovery fails the client is evicted - i.e. it erases all cached state and completes all uncompleted transactions with failure.

recovery dependence after a server failure, client (a) is dependent on client (b) for recovery if clients (a) and client (b) have such uncommitted transactions (Ta) and (Tb) respectively that (Ta) depends on (Tb). Thereby client (a) will be evicted unless client (b) participates successfully in recovery.

COS Commit on Share - a strategy to eliminate recovery dependence by ensuring that inter-client dependent transaction do not read uncommitted data.

VBR Version Based Recovery

1.2 Background

Condider a situation when a MDS server enters recovery mode on startup after failover or service restart for other reason. The clients are reconnecting and replaying their uncommitted requests to restore connection states. One or more clients missing the recovery may cause other clients to abort their transactions or even to be evicted.

The transactions of the missing clients cannot be applied. Moreover a transaction that depends on a missing one cannot be applied correctly and get aborted, the transactions depending on the aborted ones get aborted too and so on.

The COS feature attacks the source of these problems by eliminating dependent transactions. If there are no dependent uncommitted transactions to re-apply, the clients apply their requests independently without a menace of being evicted.

2 Requirements

1. Allow clients to recovery independently
2. The mechanism should be optional to allow users to choose between performance and reliability
3. Changes to the wire protocol are backward-compatible.
4. Provide compatibility for old clients

3 Functional Specification

The proposed solution is to detect inter-client dependent transactions and prevent reading of uncommitted data by doing a commit between the dependent transactions.

The solution is built around the LDLM. The following are the highlights of the solution:

- A client tracking info added to the lock object. We extend the ldlm lock policy field by a tag which identifies the client started the transaction. The point is that ldlm has no notion of client behind request unless lock is explicitly requested by client.
- After a data modification transaction completes, the corresponding **PW** lock isn't released immediately. Instead the lock is converted and preserved until transaction commit.
- The **PW** lock gets converted into special **COS** lock. The lock compatible check uses the client tracking information mentioned above to allow any **PW/PR** lock from the same client and to conflict with any **PR/PW** lock from any other clients. An early lock conflict (before the lock is converted to a COS lock) is an exception to this lock conversion rule, it is explained in more detail in the Logic section.
- Another client access is detected as a lock conflict of the client lock request and the **COS** lock. Once lock conflict happens, we forcing the transaction to commit.
- A transaction commit releases the **COS** locks associated with the transaction.

A note: data modification is protected by **EX** as well as **PW** locks. What is said in this document about **PW** locks is applied to **EX** locks too.

3.1 Relation with REP-ACK

REP-ACK and **COS** are similar. Both mechanisms achieve some levels of guarantee of recoverability of the request before unlocking the object and letting further transactions to proceed. **REP-ACK** waits a message from the client that request reply was received or transaction commit, **COS** only waits when the transaction reaches the persistent storage. **REP-ACK** relies on the client or a disk storage while **COS** trusts disk storage only.

We use **COS** when enabled and **REP-ACK** when **COS** is disabled.

3.2 Replaying of independent requests when one or more clients missing recovery

COS and **VBR** target the same Lustre release 2.0, we assume that VBR will process uncommitted requests in a system with running **COS**.

VBR may have problems with replaying uncommitted requests if there are two uncommitted requests from different clients which updates the same object and one of the clients misses the recovery. It is impossible to have such requests in **COS**-enabled filesystem because those requests are considered as dependent. Only exception is parallel dir operations. Current implementation of **VBR** updates directory object version with each file create. Its object version test will fail if some of the updates is missing in recovery. **VBR** should be adjusted to accept such requests.

3.3 Concurrent directory access

Concurrent directory access will be supported with pdirops feature which protects hash-associated part of directory with dedicated ldlm resource. COS will apply to hash-associated locks.

3.4 Managing COS

New MDS configuration parameter "commit_on_sharing" added to enable or disable COS, zero value means cos feature is disabled, any non-zero value means the opposite. It makes the COS feature to be controlled through lprocfs or lctl interfaces and cos enable status can be saved as a part of cluster configuration.

Enabling COS operation switches handling of difficult request from REP-ACK logic to COS logic immediatelly. A transaction commit is done to be sure that no saved locks might be released by client ACKs when COS is enabled.

3.5 No wire protocol change.

As explained in the section 3, we add a field the lock policy structure which is transferred through network. It extends the smallest element of a union and doesn't change size of the structure or offsets of existing fields.

4 Use Cases

4.1 A client modifies object with no changes in cache

The object is unlocked and has no uncommitted changes

- Client issues an object modification request
- The server takes a **PW** lock on the object
- The object gets modified on the server
- The PW lock gets converted into a COS lock
- The object remains **COS**-locked

4.2 The client continues with the write access to the object

The changes made to the object by the client are not yet committed to disk.

- The client issues an object modification request
- The server requests a **PW** lock on the object
- The lock is granted because the COS lock is compatible with any lock request coming from the same client.
- The object gets modified
- The client lock is released, the COS lock remains

4.3 The client continues with the read access to the object

The changes made to the object by this client are not yet committed to disk.

- The client issues a request to fetch data from an object
- The server requests a **PR** lock on the object
- The lock is granted because the COS lock is compatible with any lock request coming from the same client
- The object gets accessed
- The client lock is released, the COS lock remains

4.4 A client writes to the object recently modified by another client

The object was write locked and modified by another client, the changes are not yet committed to the disk.

- A client issues an object modification request
- The server requests a PW lock on the object
- LDLM finds the lock request conflicting with the COS lock
- LDLM calls the BAST registered with the lock
- The BAST triggers a transaction commit
- The COS lock is released by a transaction commit hook
- The lock request is granted
- The object gets modified

4.5 A client reads the object recently modified by another client

The object was write locked and modified by another client, the changes are not yet committed to the disk.

- A client issues a request to fetch data from an object
- LDLM finds the lock request conflicting with the COS lock
- LDLM calls the BAST registered with the lock
- The BAST triggers a transaction commit
- The COS lock is released by a transaction commit hook
- The lock request is granted
- The client fetches the object data

4.6 Parallel file creation in one directory

A bunch of clients create files in one directory, file name hash collisions considered as rare

- Clients take locks on hash-associated lock resources of the directory
- The locks get converted to COS locks
- Subsequent file creations are compatible with the COS locks and cause no transaction commits

4.7 Enable COS

COS was disabled

- An MDS server saves locks and sends replies with LNET_ACK_REQ flag set
- A user issues lctl config_param command
  lctl set_param mdt.*.commit_on_sharing=1
- lctl communicates with the MDS server using procfs interface
- local MDS servers change their cos status
- local MDS servers commits all transactions
- saved locks are released by the transaction commits

4.8 Disable COS

COS was enabled

- An MDS has converted PW locks into COS ones and saved them, the reply is sent with LNET_ACK_NOREQ flag set
- A user issues lctl config_param command
  lctl set_param mdt.*.commit_on_sharing=0
- lctl communicates with the local MDS servers using procfs interface
- local MDS servers change their cos status
- lock conflict triggers transaction commit (the BAST logic is not affected by COS status change)
- saved locks are be released by the transaction commit

5 Logic Specification

5.1 Storing client id within ldlm_lock object

The ldlm_policy_data_t structure (part of ldlm_lock object) is extended to store connection cookie. The field is initialized at lock request creation and is used to check whether a COS lock and a lock request are compatible (see below).

5.2 COS lock

COS lock objects have a new lock mode, LCK_COS.

COS locks live only on resource's granted queue, as result of PW locks conversion. COS locks cannot be requested. COS locks cannot be on resource's waiting queue.

The ldlm_inodebits_compat_queue() function is changed to compare request locks and COS locks.

COS lock compatibility table:

Lock request	COS lock
PR/same client	compatible
PW/same client	compatible
PR/another client	no
PW/another client	no

5.3 Triggering transaction commit

We define a BAST method for any PW lock taken by server. However, the BAST can be called before the PW lock is converted to COS lock. The transaction might not be started and there is nothing to commit yet. The BAST should perform a commit only if the lock has been converted to COS lock already.

If the BAST is called before the lock gets converted, the lock conflict doesn't necessary mean that we get transaction dependency, PW/PR locks from one client may generate a lock conflict as well. We have to do additional check over the lock and the lock request to find whether there is a transaction dependency or not.

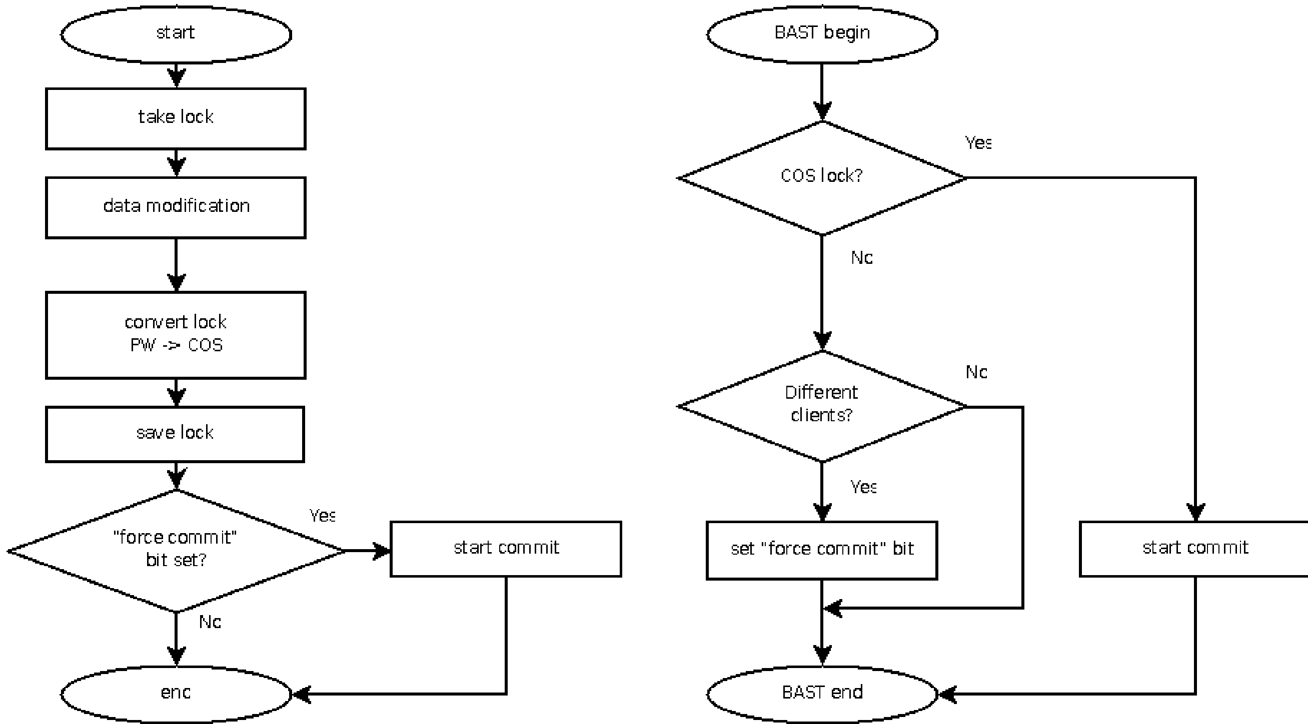
The check is like the COS compatibility check defined above, but we have no COS lock yet:

Lock request	existing PW lock
PR/same client	independent
PW/same client	independent
PR/another client	dependent
PW/another client	dependent

If it is found that transactions are dependent, we want the data modification transaction to be committed synchronously.

We introduce new "force commit" bit in the lock flags. The bit is set when transaction dependency is detected but the lock isn't yet converted to a COS one. The bit keeps the information about a need of commit until the data modification is completed and the transaction can be committed.

The key modifications to the data modification routine and the blocking AST are illustrated by the following flowchart diagrams:



The commits are are done asynchronously, by another thread. We only ask it to commit the transactions. After the thread commits the transactions it calls commit hook where saved locks get released.

However commit start could be issued more then once for one transaction. There is number of reasons for that: many COS locks for one resource and thereby many BAST messages and many lock resources involved into the transaction. Some of the sync starts can come late when the transaction has been committed already. It may have a negative effect of forcing the next transaction to commit. The negative effect isn't measured yet and considered as low one for now.

5.4 Server reply and client ACK

If we don't use ACK for releasing the locks and the request, we don't set the LNET_ACK_REQ flag with the server reply.

5.5 Saving COS lock until commit

REP-ACK already has an infrastructure (ptlrpc_save_lock) to save and keep locks until client ACK or transaction commit. We reuse that whole infrastructure for COS. The COS lock will be kept until transaction commit because we don't require client ACK to be sent on the server reply.

5.6 (Not) saving PR locks

REP-ACK mechanism is used for read (**PR**) locks. There is no reason do that neither for **REP-ACK** nor **COS**. Both mechanisms should not save PR locks but decref them.

6 State Management

7 Alternatives

7.1 Lazy commit

We don't force commit immediately after inter-client dependency is found, but have coming client to wait a commit triggered by timeout or other reasons.

8 Focus of Inspection