High Level Design

Mike Pershin, Alexander "Zam" Zarochentsev

May 18, 2008

1 Introduction

Commit On Sharing technique helps to recover more clients using version-based recovery. It does commit if any resource has uncommitted changes from more than one node. Therefore all replays on clients will not depend on other clients and clients may recover cleanly even if several clients are missed.

2 Requirements

1. A server should:
   (a) determine cases when different clients are going to use the same resource (sharing)
   (b) check if there any uncommitted data from other clients about this resource
   (c) perform commit of all changes for shared resource if needed
2. COS feature should be optional and manageable through some administrative tool like lctl.

3 Functional Specification

3.1 Definitions

COS commit on sharing

VBR version-based recovery

version {server boot sequence number, transno} pair

last_committed last committed transno

uncommitted inode an inode that has uncommitted changes

shared object two or more nodes have altered the object and those changes not yet committed to disk.

operations any of the following meta-data operation: getattr, setattr, insert name, delete name, link, unlink, rename

dependent operations two operations are dependent if they are issued by different clients and 2nd one can not be executed until 1st one is executed, otherwise they are **independent**. We consider only cases when both operations were successful and can be replayed.

worker thread a thread which handles client requests

3.2 The problem

In VBR, operations done to an object, get sequential version numbers. To have a full recovery, there should be no gaps in request versions. Suppose the operations are done by several clients and one or more clients miss the reconnect window. Highly probably we have non-trivial gaps version in the sequence and the recovery process fails.

3.3 The solution

The solution is to eliminate dependent operations originated from different clients. If a set of requests to replay have no inter-client dependencies, the recovery would survive if one or more clients are missing.

To eliminate that dependencies, the COS code forces transaction sync when inter-client dependencies are detected.

This HLD suggests to use the simple dependency tracking. With that, COS becomes a simple feature to allow only one client to have uncommitted changes for one object. Unfortunately parallel activity like parallel creates in one directory is not possible with simple COS.

3.4 Useful optimization

Instead of inserting transaction synchronous commit between two dependent operations (Op1 and Op2) right at the moment when the dependency is detected, it is useful to do synchronous commit after O2 is done. The result of O2 is sent to the client after the changes reached the disk, so we still can't have requests for replaying from different clients targeting one object.

The optimization may reduce number of synchronous commits and simplifies the implementation as it is shown in the logic specification section.

3.5 Managing COS

COS is controlled by Lustre lproc tunable, its value is stored as a part of Lustre configuration.

3.6 Relation with REP-ACK

When enabled, **COS** makes **REP-ACK** not needed.

4 Use Cases

4.1 2 clients alter own object each

- COS shouldn't find any dependency and thereby no commit would be forced. COS is silent in this case.

4.2 Independent operations from different client

- 2 clients create many files in the same directory.
- simple COS finds false dependencies between operations and transaction commit is forced.

4.3 Dependent operations from different clients.

- 2 clients create and delete files in the same directory
- simple COS makes the current transaction commit synchronous.

5 Logic Specification

5.1 Storing object's change history

dependency objects

- For each file or dir object with not yet committed changes we have a dependency object.
- A dependency object contains UUID of the latest client altered the object and the Lustre transaction number.

hash table

The dependency objects are organized in a hash table, indexed by FIDs.

garbage collection

Outdated (its transno $<$ last committed transno) dependency objects are removed from the hash table. it is done in hash bucket scans (i.e. in any hash table lookups) and periodically by a GC thread.

5.2 Checking dependency

- The hash table is searched for the corresponding (with the same ino as the object has) dependency object.
- If no not-outdated dependency object found we assume no dependency and insert new dependency object or update the outdated one with new trasno.
- If not-outdated dependency object found the dependency check is done by comparing client UUID stored in the dependency object and current client UUID. There is a dependency if the UUIDs are different.
- if there is a dependency the transaction is forced to commit after the operation is done by setting sync-on-close bit in the transaction handle. The dependency object's transno, client UUID fields get updated.

5.3 Rep_ACKs and COS

Current Rep-ACK locking mechanism is not well suitable though doing the same. It should be changed a bit. Rep-ACK locks objects in many cases even when it is not needed actually for COS therefore the following should be done:

1. lock should be drop right after operation and no Rep-ACK should be sent;
2. COS eliminates the need for Rep-ACKs, therefore in COS feature is turned on then Rep-ACK functionality should be turned off, that can be done avoiding invoking the `ptlrpc_save_lock()`

5.4 Compatibility

Commit on sharing don't have compatibility issues between 1.6 and 1.8 but should be updated for HEAD-based code later.

6 State Management

6.1 Locking

The hash table uses one spinlock per hash bucket.

7 Alternatives

7.1 Complex COS dependency tracking

to allow parallel operations not to commit transactions if they proved independent as it defined at the beginning of the document.

7.2 Lazy commit

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.

7.3 alternative method for storing dependency tracking info

embedding the info into struct ldiskfs_inode_info

8 Focus of Inspection