HLD for Gaps Handling Simplification

Huang Hua

Feb 17, 2006

Contents

1 Introduction

This document describes the issues related to the simplification of gaps handling in Lustre. Gap is an issue in Lustre recovery time. To handle gaps correctly and with simplicity is important. Some modeling and verification about gaps is described in **??**.

1.1 What is a GAP?

First of all, gap is an issue in the recovery phase. Consider the following example:

1. A cluster is running Lustre. There are clients, MDS, and OST. Everything is going well;
2. At some time, the clients are creating bunch of files or doing other activities, and then the MDS fails;
3. After the fail-over MDS is up or the failed MDS is up again, recovery starts from here:
   (a) The MDS is up and starts its own recovery. This means mostly that it synchronizes its own objects with OST to avoid cases when OST has some object but MDS has no inode for it what would cause inability to open that object.
   (b) Client re-connects to MDS and starts its own recovery. This means mostly that this client replays all RPCs which did not fit into last transno on MDS. Client knows that because its last transno received from MDS before failure is bigger than last committed transno on MDS after failure. Client gets the latest committed transno from MDS when MDS committed these transactions. The latest committed transno is included in all replies from MDS. Client tries to replay all its RPCs to MDS in correct order, and MDS handles these RPCs in an increasing order. Currently, the order is achieved by taking DLM lock: client takes DLM lock before sending replay RPCs to MDS and when MDS is sure that it replied that RPC - MDS releases the lock, so that MDS knows that there is NO parallel replies and order to be preserved.
   (c) Now suppose that client A can't replay RPCs due to its crashing or network issues, and all its transactions with transno [N,K] were lost. These transactions will never be replayed ever. But other clients are still trying to replaying their transactions, saying [K+1,L]. In this case, transactions

[N,K] is called a *gap*. A gap can contain transactions that did not receive commit replies and the results of those transactions may have been used by other transactions.

4. The current schema to handle gap is as following:

   (a) The recovery handling notices a gap because not all requests are sent for replay. MDS notices this because it gets timeout while waiting for the next replay.

   (b) So we try to allow recovery not to evict ALL clients, but ONLY those clients that have not finished replay because they may depend on the gap. We know this because clients send replay RPCs with special flag to indicate its finishing of replay.

   (c) Those evicted clients will discard all the RPCs which were planned to replay in the previous steps because these replay requests may depend on the gap. MDS never detects whether successive replay RPCs really depend on the gap, but just evict them. So these operations/transactions are discarded and seems that they never happened. These evicted clients then connect to MDS again, and resend those RPCs which haven't got replies once again from scratch, and start new operations.

1.2 What is FID?

FID is used to uniquely identify an object in Lustre. FID contains some components, such as FID sequence, FID number. These components have some useful and particular information about objects. Please refer to the FID HLD. FID is used by clients to communicate with MDT and OST.

Clients somewhat maintains FID. The FID sequence is an increasing number, and has different value every time the client start a new connection to MDT and/or OST.

1.3 Disadvantages in current gaps handling

The current gaps handling during recovery time has the following disadvantages:

- It may evict some clients which do not depend on the gap. So this will increase the time overhead.
- It may discard operations which do not depend on the gap. So this may cause some applications not to survive from the recovery.

2 Requirements

Now the FID is an universal component of Lustre, and is the basis for many other features and components. The main requirements for gaps handling are:

- Simplify the recovery process for handling gaps;
- Speedup the recovery when there is a gap;
- Try to recovery as much as possible, and without any inconsistency;
- Fulfil the promise that Lustre survives single point of failure;
- Use FIDs to simplify recovery, particularly in gaps handling.

3 Functional specification

In order to meet the requirements, the essential idea here is to detect the dependency between transactions and/or between clients. Those transactions which do not depend on a gap should be replayed, and those clients which do not have transactions depend on the gap should not be evicted. By these means, we may speedup the recovery phase, save as much metadata as possible.

3.1 Detecting and tracking dependencies

When the MDS is handling RPC replays during recovery, the best case is that all clients reconnect to MDS and finish replaying, then finish resending, and then start new transactions. If some clients can not replay, gaps may arise.

We know that some clients have finished replaying because they have sent RPC requests with special flag to indicate that. We also know that there is a gap because some requests are not sent for replay. The previous schema for gaps handling is to evict all the clients which have not finised replaying because their remaining transactions may depend on the gap. We do not detect whether other clients really depend on the gap, just assume that all of the un-finished replay may depend on the gap.

We need a smart way to detect and track the dependencies between clients and/or between transactions. The way should be simple and efficient, without much overhead. It should give us much advantages during recovery when there are gaps.

There are two kinds of dependenies:

- Dependency between clients. Client A depending on client B means that client A will do some decisions based on the metadata/data which has beed modified by client B. This dependency is client node level.

- Dependency between transactions. Transaction X depending on transaction Y means that transaction X will read/write some metadata/data which has been modified by transaction Y. This dependency is object level. It has a smaller granularity that the above one.

3.2 Gaps handling based on dependency

When the recovery handling notices a gap, it detects whether other clients depend on the failed client which causes the gap. We have two different ways to handle gaps.

- Dependency between clients. If some clients do not depend on the gap, MDS continues with the replaying from these clients. Only the clients which depend on the gap should be evicted.
- Dependency between transactions. If the replaying transactions after the gap do not depend on the transactions in the gap, MDS should accept these replays. Those transactions which depend on the transactions in the gap will be discarded. Those transactions which depend on discarded transactions will be discarded too. Those transactions which depend on accepted transactions will be accepted too.

4 Use cases

Use cases based on different dependencies are different.

4.1 Dependency between clients

Based on this dependency, we have the following gaps handling schma:

1. MDS is handling replays from clients;
2. Some requests are not sent for replay and connection is timeout. MDS notices there is a gap.
3. Check all the remaining active clients, evict those clients which depend on the timeout client which caused the gap.
4. Evict all the clients which depend on those clients which has already been evicted.
5. Repeat step 4 until there are no more clients depending on timeout or evicted clients;
6. If there are any clients remaining, continue handle replays from these clients, and continue recovery as before.

4.2 Dependency between transactions

1. MDS is handling replays from clients;
2. Check the dependency for the replay transaction named X;
   (a) If this transaction X depends on transaction Y and Y has been accepted, accept X;
   (b) If this transaction X depends on transaction Z and Z has been discarded, discard Z;
3. Repeat step 2 until there are no more replays, and continue recovery as before.

5 Logic specification

5.1 Overview

This section will describe how to detect and track the dependencies between clients and transactions. We will also discuss their pros and cons.

5.2 FIDs rules

According to the FID HLD, FIDs have the following rules (not limited to):

- Objects that have the same FID sequence only be stored on the same MDT.
- Every time the client connects to MDT, it will get new FID sequence, and will only create objects with this new sequence. Every pair of client and MDT have a different FID sequence.
- When a client re-connects for recovery, it will use the same FID sequence as before. When the recovery fails and connects again, new FID sequence is used.

These rules may be useful to the gaps handling. We can use these special characteristics of FIDs to help us make dependency decision.

5.3 Dependency between transactions

Dependency between transactions may be detected by MDS and tracked by clients. We may detect the dependency at the object level:

- if transaction X modifies object O & P; We record X in object O & P (X is the last transaction which modifies P & Q);

- if transaction Y operates on object M & N, then Y does not depend on X; Y may depend on previous transactions which modify M & N;
- if transaction Z operates on P & Q (Z and X both operate on the same P), then Z depends on X (Z also depends on some transaction which has operated on Q); Now we record Z in object P & Q(Z is the last transaction which modifies P & Q).
- When an object is new, the transaction which creates this object depends on transaction ZERO or depends on itself. Both are OK.

We assume that Z = Y + 1, and Y = X + 1 in above example. So "Transaction B depends on transaction A" means that if and only if transaction A has been committed, transaction B can be committed. This dependency only consider the integrity and consistency of file system, without considering the application using file sytem. In the above example, if the transaction X is lost, we may commit transaction Y, but we MUST NOT commit transaction Z.

When a transaction/request is accepted by MDT during normal operations, compute the transactions it depends on, and send these transactions' numbers back to clients in RPC reply. A transaction which operates on some objects may depend on multiple transactions which operates these objects some times ago. These dependent transaction numbers are sent back to clients in RPC reply, and the clients should record them in its requests. When these requests are replayed in recovery, the dependent transaction numbers are sent to MDS along this request, and MDS can decide how to handle it.

The last transaction which operates on a object is recorded in this object, so later transaction can know this previous transaction which operates on this object.

So this kind of dependency is detected by MDT and kept (tracked) by clients. These dependencies are never written to persistent storage, so it has a small overhead.

When the MDS is handling replay from clients, it waits for the next transaction from clients. If some clients crashed, and replay were not sent, MDS will get a timeout error. It marks this transaction as "not accepted" or "discarded", evicts this client, and goes on to handle the next one. Only those transactions which depend on accepted ones will be accepted. This schema may take a longer time than the previous one, but it will save more metadata than just evicting all un-finished clients.

5.4 Dependency between clients

Dependency between clients may be detected and tracked in a similar way, except that in a client level:

- if a transaction from client A modifies object O & P; We record A in object P & Q (A is the last client which modifies P & Q);

- if a transaction from client B operates on object M & N, and client A has not modified M & N since last commit, then B does not depend on A; B may depend on other clients which modify M & N;
- if a transaction X from client C operates on P & Q, then C depends on A (C alsp depends on some clients which has operated on Q); Now we record C in object P (C is the last client which modifies P); we say trasaction X from C depends on A.

By using the same technique, we save the client information in every object, and return the dependency information back to client in reply. When some client causes a gap, MDS just evict it, and then continue to handle next replay. If this replay reqeust depends on evicted client, evict the client this replay from. Otherwise accept this replay as a good one. Repeat this step until there is no more replay requests to handle.

Please note that this dependency is somewhat a probable dependency between clients. We do not check the precise/exact dependency between clients. At the same time, there might be cyclic dependency between clients. So we may periodically force to commit all transactions and clear the dependency. In a random time interval, there will not be too much replay requests to handle.

6 State management

6.1 State invariants

1. Object in memory

   We want to record in the object representing structure in memory the last transaction which has modified this object, or the last client which has modified the object. So the object representing structure in memory need some extra members to remember the last transaction number or the client number.

2. RPC request & reply

   We want the clients to remember their requests' dependency relationship, so these dependency relationship may be used to help recovery. MDS sends these dependencies back to clients in RPC reply. So the request and reply RPC need extra members to hold thses information.

6.2 Scalability & performance

The gaps handling technique described by this document will save more metadata during than before, because it does not evict all un-finished client, but only discard those transactions which depend on "discarded" transactions.

Because we do not evict those clients which do not depend on the gap during recovery time, better scalability and performance may be achieved. Recovery may be completed in a shorter time period.

6.3 Recovery changes

This is the document which describes gaps handing in recovery. Other issues about recovery may be discussed in other HLDs.

7 Alternatives

The most simple algorithm is to evict all those clients except for those which have finished replaying, and it maybe can give out the best application integrity.

8 Focus for inspections

1. Are there any problem in detecting and tracking dependency between transactions?
2. It seems that I have not used any FIDs proprietary feature. What can FIDs help?
3. Are there any potential problem?