Composite Layout with Erasure Coding High Level Design

Introduction

The Erasure Coding High Level Design describes details of how the EC feature may be implemented, including the user interfaces, how files with EC feature interact with client-IO layer, the RPC interchange between the client and servers.

Design Overview

There are three main components to the EC design:

* The user-space interfaces for Lustre-specific command line tools and user library application programming interfaces (APIs)
* Changes to the client IO (CLIO) layer IO engine and RPC interfaces to the MDS for accessing and writing composite file layouts
* Changes to the MDS server to create, modify, and delete composite files

The design is structured in a top-down manner, starting with the command line interfaces that users, then the user library APIs, the client-side kernel changes, RPCs for creating and modifying composite files, and server-side changes.

User Space Interfaces

The lfs(1) command line interface will be extended to understand and manipulate EC files and their component layouts. User will use it to create files with EC layout, show the layout of existing files, and set default EC layout template on directories which can be inherited by new files and sub-directories created therein.

**lfs setstripe**

The *lfs setstripe* command creates a new file with the specified layout parameters, or sets the default layout template for a directory.

lfs setstripe {--component-end|-E end1} [comp1\_options] **{-L erasure\_code|ec} [--ec\_data\_count number\_of\_data\_devices ] {--ec\_code\_count number\_of\_parity\_devices}** {--component-end|-E end2 } [comp2\_options] [**{-L erasure\_code|ec} ...**] {filename|directory}

lfs setstripe **--erasure-del** filename # delete all EC components

lfs setstripe --comp-del -I [ec\_comp\_ID] filename # delete an single EC component

Since this is the primary command line interface for users to create new files with Lustre specific layouts, there are a number of existing options can be used. Adding erasure coding specific options allows the same code to create erasure coding enabled composite layout without duplicating a large number of options. The existing --component-end arguments are used to specify the code component corresponding the data component whose extent ends with *end1* , and **--erasure-code** arguments following the data component arguments are to specify the parameters of coding component associated with the corresponding data component. **--ec\_data\_count** is used to specify the number of data stripes used to compute the erasure code out of them, without setting this option, all of the associated data component stripe devices will be used for the code calculation. **--ec\_code\_count** specifies the number of code stripes to be used; And the act of encoding takes *k* data devices, calculates *m* code devices. The following **-i/-o/-p** options can be used to specify which OSTs to be used as the code devices.

As an example:

$ lfs setstripe -E 4M -c 4 -L erasure\_code -m 2 -E eof -c 32 -L ec --ec\_data\_count 6 --ec\_code\_count 2 /mnt/lustre/file

Every code component uses the same stripe size as its corresponding data component’s stripe size, and the parity computation only involves its corresponding component’s data, if the tailing data of the component are not aligned for the *k* group of stripes, a memory filled with 0 padding will be used for the parity computation.

This creates a file with two data components and two code components

**lfs mirror resync [-y] <EC\_file\_name>**

This command is used to resynchronize an out-of-sync erasure-coding file. If there are no stale code components in the EC file and no *-y* argument is used, this command does nothing. Otherwise, this command will first hold the exclusive lease of the file, and then the file will be resynchronized. During the resynchronization, the file data will be read and generate the parity code, then the parity codes will be written to the OST objects of the code components. After the parity codes are synchronized, this command will change the layout to mark the code component as uptodate and release the lease of the file.

**On-disk/wire structure changes for Erasure Coding layout**

* layout header

struct lov\_comp\_md\_v1 {

 \_\_u32 lcm\_magic;

 \_\_u32 lcm\_size;

 \_\_u32 lcm\_layout\_gen;

 \_\_u16 lcm\_flags;

 \_\_u16 lcm\_entry\_count;

 \_\_u16 lcm\_mirror\_count;

 \_\_u16 lcm\_ec\_count; -> code component count

 \_\_u16 lcm\_padding1[2];

 \_\_u64 lcm\_padding2;

 struct lov\_comp\_md\_entry\_v1 lcm\_entries[0];

};

* code component entry header

struct lov\_comp\_md\_entry\_v1 {

 \_\_u32 lcme\_id;

 \_\_u32 lcme\_flags;

 struct lu\_extent lcme\_extent;

 \_\_u32 lcme\_offset;

 \_\_u32 lcme\_size;

 \_\_u32 lcme\_layout\_gen;

+ \_\_u8 lcme\_dstripe\_count; -> data stripe count used in EC

+ \_\_u8 lcme\_cstripe\_count; -> code stripe count, m value

 \_\_u16 lcme\_padding\_1;

 \_\_u64 lcme\_timestamp;

};

* code component entry blob

struct lov\_mds\_md\_v4 {

 \_\_u32 lmm\_magic; -> LOV\_MAGIC\_EC

 \_\_u32 lmm\_pattern;

 struct ost\_id lmm\_oi;

 \_\_u32 lmm\_stripe\_size;

 \_\_u16 lmm\_stripe\_count; // total stripe device count

 \_\_u16 lmm\_layout\_gen;

char lmm\_pool\_name[LOV\_MAXPOOLNAME + 1];

 struct lov\_ost\_data\_v1 lmm\_objects[0];

};

**Memory structure changes for Erasure Coding layout**

* layout header

struct lov\_stripe\_md {

...

 u32 lsm\_magic; -> LOV\_MAGIC\_COMP\_EC

...

+ u16 lsm\_ec\_count;

struct lov\_stirpe\_md\_entry \*lsm\_entries[];

};

* code component entry

struct lov\_stripe\_md\_entry {

 ...

 u32 lsme\_magic; -> LOV\_MAGIC\_EC

+ u8 lsme\_dstripe\_count;

+ u8 lsme\_cstipe\_count;

...

};

The code component has the same extent range as its associated data component, and the code component descriptor is posited after data components.



Layout Component Creation

Creating layout component on demand still uses layout intent RPC to notify the MDS to prepare the file’s components as what PFL/FLR files do.

Write to EC file

The first phase of this project is aim to build the infrastructure to support erasure coding in Lustre. To simplify the implementation, this design only supports delayed parity calculating, i.e., the data write does not generate parity code at the same time, it will only mark the code components stale, and write continues on the striped data components. After the file is closed, administrator can use an external tool to generate and write the parity code. Marking code component stale will be recorded in the Lustre ChangeLog so that the resync tool or external policy engine can detect and resync the code components.

With delayed write, the write intent RPC handler in MDS will instantiate destination data and code component, and mark the code component STALE. The layout generation will be increased to notify the other clients to clean up their cache. And these code components will stay stale, and read cannot use them to rebuild missing data until they got resynced later.

Map file offset to code offset

Each stripe of parity code is calculated by a polynomial taking several (say k) stripes of data as its variables, so that every k bytes of data in the file corresponds to a byte of code in one of the corresponding component code stripes.

Suppose the data of offset *off* locates in component *m*, we use km stripes for code generation in component *m*, and component *m* starts at *compm.start,* the stripe size of this component is *sm*, so the code offset *code\_off* in the code object corresponding to the data at *off* can be calculated as follows:

Let *offm = off – compm.start*

And *code\_off = offm/(km \* sm) \* sm + offm % sm*

Code Resynchronization

After an EC file is written, its code component is not valid to reconstruct normal data if data OST is not available. We need a mechanism to synchronize the code, i.e., compute the parity code from normal data and write the code to the code component, then update the layout to indicate that the parity in the code component is valid so that read can leverage them to recover data from possibly failed OSTs.

Read

When reading from an EC file, if the read I/O encountered failed OSTs, the IO framework detects the error and specifically LOV layer will manage the page to read parity code from the code component and re-generate the missed data page for the failed OSTs, so that client can read without noticing OSTs’ unavailable.

Extent lock expand

Depends on the number of code fragments parameter in the erasure-coding, to calculate a data value in certain a file offset, all the data stripes in the relevant code fragments need to be read, which could possibly involve reading the data not in the current IO request range, so that means we also need to take read lock covering that part of data. In the case of an OST failure, then read will be restarted and we would take the extent lock covering the relevant data and code fragments.

IO framework for read

the IO framework retry mechanism is triggered, and the retry created another EC I/O, which is meant to read data stripes of available OSTs and parity code OSTs, and calculate data belonging to failed OSTs.



Page management in LOV

To shield user space tools and LLITE layer from aware of the parity code directly, LOV layer will take charge of the page management of the parity code belonging to the code components. LOV will acquire CR extent lock of the requested code OST object, allocate cl\_page associated with the parity OST object, and since the code OST object is obscured from LLITE layer, so that it’s LOV layer’s responsibility to maintain the code pages. These pages can be attached to the code extent lock and/or maintained in an LRU facility. When the lock got canceled, the parity cache can be discarded accordingly.

Requirements

Erasure Code Library

We’d choose an existing library supporting erasure code encode and decode. Intel Intelligent Storage Acceleration Library (ISA-L) (<https://github.com/01org/isa-l>) supports fast block Reed-Solomon type erasure codes for any encode/decode matrix in GF(28), and we can leverage for the parity code generating and data restore.

User Space Tools

A Lustre user space tool is needed to define and set parity components for a file. We can reuse *lfs mirror* *create* to serve this purpose.

Another tool is needed to generate the erasure code for the changed data block and update the parity code. We’d reuse *lfs mirror resync* to check internally whether the component to resync is a mirror or a EC.

Erasure Coded File Write

Erasure coded file writes will mark the corresponding parity component stale. After the file is closed, resync tool can be used to clear the write extent list and generate/update erase code for corresponding parity component.

Erasure Coded File Read

The Lustre client will do normal reads from the RAID-0 data component, unless there is an OST failure or other error reading from a data stripe, a read recovery will be started, reading erasure code data from parity components and reconstruct the data for the failed OST.