Layout Enhancement Solution Architecture

1. Introduction

The following solution architecture applies to the Layout Enhancement project within the Technical Proposal by High Performance Data Division of Intel for OpenSFS Contract SFS-DEV-003 signed Friday 23rd August, 2013.

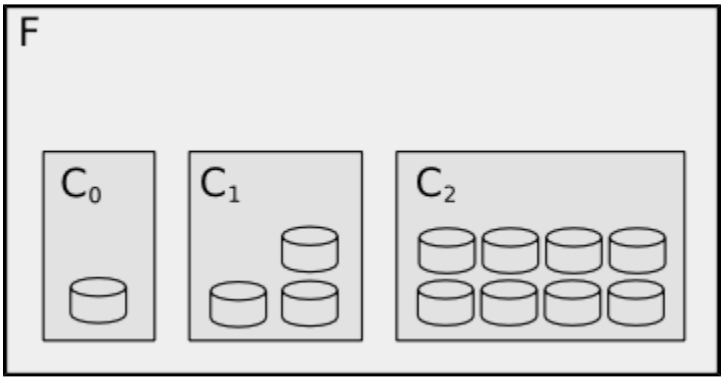
In the Lustre* file system the data of a file is striped over one or more objects each residing on an OST. The *layout* of a file is an attribute of the file which describes the mapping of file data ranges to object data ranges. The layout is stored on the MDT as a trusted extended attribute (`trusted.lov`) of the file and are sent to clients as needed. The layout of a file is often simply referred to as its *striping*, since in the current (2.5) implementation of Lustre only non-redundant striped (RAID0) layouts are permitted. This project will design enhancements to the representation and handling of layouts to support features such as Data On MDT, File-Level Replication, live data migration (via File-Level Replication), and RAID1/5/6 or erasure coding. The Layout Enhancement (LE) project is therefore a prerequisite for the Data on MDS and File-Level Replication projects, described in separate documents.

2. Solution Requirements

There are several demands on the Layout Enhancement project, each with its own set of requirements.

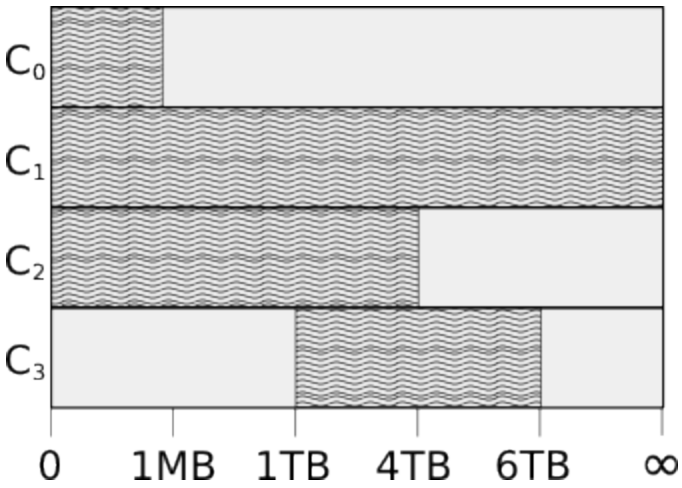
2.1 Layouts for File-Level Replication

File-level replication will use mirroring to offer increased availability and resilience of data on a Lustre file system. File-Level Replication is discussed in more detail in the File-Level Replication Solution Architecture. The implementation of File-Level Replication requires a new *composite layout type* that comprises several *simple* (non-composite) layouts designating mirrors of the file data. It generalizes RAID0+1 to allow heterogeneous stripe sizes and counts among mirrors and to allow for the possibility of partial mirroring schemes, potentially with each replica on different OST storage pools with different performance and capacity characteristics. The design will specify the interaction of layouts for file-level replication with existing code that interprets or manipulates layouts (HSM, LOD, LOV, utilities, layout swapping, ...). Figure 1 below depicts a 3-way replicated file *F* with components C_0, C_1, and C_2 having 1, 3 and 8 stripes respectively.



2.2 Extent Based Layouts

Extent based layouts permit different layouts to be used in different extents of a file. They may be used to set progressively wider striping as a file grows in size, to prevent inconsistent out of space errors as individual OSTs become full, and to enable incremental migration, replication, and HSM restore. Figure 2 below depicts a file with four different layouts (C_0 through C_3) each with its own extent to cover both overlapping and non-overlapping parts of the file data.



2.3 RAID-1/5/6/10 and Other Algorithmic Layouts

These anticipate future developments to enable more space-efficient replication techniques. The design will discuss how these layouts can be expressed with simple extensions to the existing layout code. While RAID-1 and RAID-10 layouts offer degrees of replication they are not to be confused with the layouts for file-level replication described above. They are fixed layouts similar to the existing RAID-0 layouts currently used by Lustre. Compared to composite layouts they are simpler and more compact but they are also less expressive and less flexible. RAID1 and RAID-10 layouts may be converted to composite layouts. Composite layouts whose components all have the same stripe size and stripe count may be expressed as RAID1 and RAID10 layouts. Full read/write support for files with RAID-3/5/6 layouts is beyond the scope of this design. Instead we will briefly outline an "offline parity" access mode for files with these layout types.

2.4 Layouts Based on the CRUSH Algorithm

After investigating the CRUSH layout model it has been determined that the complexity of implementing a CRUSH-based solution within the current Lustre code is beyond the scope of this effort. Potential layouts could be proposed but we are not confident if they would be correct or complete by the time of implementation. We are confident however, that the layout proposed in this document is flexible and could allow for CRUSH-style algorithms in future.

2.5 Compact Layouts for Widely Striped Files

Existing (RAID-0) layout formats use an explicit array of object identifiers to map each stripe index to a specific OST object. When using FIDs alone to identify objects this approach requires 16 bytes per stripe. The current implementation packs an OST index together with the object identifier and needs 24 bytes per stripe. The allocation of memory buffers to transmit, receive, and handle these layouts for very widely striped files (over 160 stripes) can be costly. We will discuss a compact layout based on a bitmap of OST indices which reduces memory consumption for widely stripe files by a factor of 192 for the current maximally-striped layout of 2000 stripes.

2.6 Handling of Large Layouts

Issues with the handling of large layouts will be discussed and various solutions to these issues will be considered. In particular, for very large layouts it is desirable that the retrieval of the layout be separated from the initial open RPC in order to avoid the need for large request/reply buffer allocation. Instead, a bulk RDMA transfer could be used to fetch the layout from the MDT.

3 Use Cases

Since the Layout Enhancement Design itself is focused on providing an infrastructure to describe a flexible layout for complex files, the use cases are described in terms of potential uses of the enhanced layouts. In some cases, there are projects underway to implement these use cases, but in other cases these are just potential uses that may be implemented in the future.

3.1 File data availability

A user wants to change an existing file to be robust against temporary or permanent OST failure. This requires that all of the file data be stored on more than one OST, so that it can be read if an OST is overloaded or permanently offline due to failure.

The File-Level Replication (FLR) project will use a composite layout with two or more overlapping extents to keep file data available in the face of OST failure. Due to the use of per-file layouts rather than per-OST replication, it is possible to selectively replicate files on an as-needed basis, such as 1-in-12 or 1-in-24 application checkpoints would have two-way replication and 1-in-72 checkpoints would have three-way replication. This allows users/applications to balance the replication and availability needs against space and bandwidth constraints, and is not an all-or-nothing decision.

In the FLR Phase 2 it will be possible to create replicated files from the beginning. In FLR Phase 1 and later it will also be able to add replication to an existing ated file.

User tools as described in the FLR project could use the following layout operations to add redundancy to non-replicated files:

1. A file's RAID-0 layout is converted to a composite layout whose sole component is the previous layout.
2. A temporary RAID-0 file is created to hold the new replica data (this is not connected to the layout and will be deleted in case of failure).
3. The file data is copied to the temporary replica file.
4. The new replica is merged into the composite layout as a new component, resulting in a compound layout with overlapping extents.

3.2 File data read performance

A user wants to have high-performance read access to a file from a large number of clients. This requires that the same data be stored on a large number of OSTs, so that it can be read in parallel at an aggregate bandwidth larger than what is available from a single OSS.

Similar to **File Data Availability**, the File-Level Replication project will use a composite layout with overlapping extents. The number of replicated extents can be determined by the required read bandwidth and available OSS nodes. Having a large number of replicas on a file only makes sense for read-only files.

3.3 Reducing redundancy for old files

If a user no longer needs a high degree of replication on a file, either because has been backed up to a separate archive, because a high read bandwidth is no longer required, or because of quota limitations, it is possible to remove one or more replicas from a file.

User tools as described in the FLR project could use the following layout operations to remove redundancy from replicated files:

1. A replicated file is destroyed, each layout component is destroyed.
2. A replicated file has one of the component replicas split from the file and it is destroyed.

3.4 Improved small file performance

A user want to store small files so that they can be read/written efficiently. Using a layout that specifies that the file data is stored on the MDT, as described in the Data On MDT Solution Architecture (DOM) allows accessing the file data with fewer disk IO operations and fewer RPCs. The DOM layout allows specifying that the data is stored in the MDT inode.

3.5 Improved performance for the start of a file

Some applications need improved performance to the start of the file, for operations such as determining the file type, accessing file metadata, or accessing a file index header, etc. This can also be useful in conjunction with HSM, where the start of the file is resident on the MDT, but the rest of the file is archived on tape. This can be achieved by using a Data On MDT component with an extent covering the first stripe_size of file data, and having an OST-based component with an extent covering the extent `[0, )` or `[stripe_size, )`, depending whether the first chunk of data will be replicated to the OST object(s) or not.

3.6 Increased bandwidth and capacity for larger files

A user wants to optimize small files with a single OST stripe, and large files with many OST stripes, without having to explicitly manage this on a per-file or per-directory basis. This could be implemented with a compound layout with multiple layouts in non-overlapping extents. As a file grows in size, progressively wider striping is used for file data in order to give the file access to more OST storage capacity and IO bandwidth.

For example, a new file could be created with a single stripe for the extent `[0, 32MB)`. If a file grows beyond 32MB in size, a new component layout would be created for the extent `[32MB, 1GB)` with 8 stripes. Should the file grow beyond 1GB in size, a third component layout would be created for the extent `[1GB, )` that is striped over all of the available OSTs.

3.7 Handling out-of-space on an OST

If an application is writing to a file it is possible that one of the OSTs runs out of space, while other OSTs have a larger amount of free space (e.g. if new OSTs were added and had much more free space, or if a very large file was created with a single stripe). This could be implemented by converting the existing RAID-0 layout to have an extent from `[0, file_size)` and then creating a new RAID-0 layout as a separate extent `[file_size, )` for the end of the file.

3.8 Transparent migration of files between OSTs

A file needs to be migrated between two OSTs. This may be needed in order to evacuate an OST that is failing or scheduled for hardware replacement, with a file that is in use by a long-running application.

The file can be converted from a non-replicated file to a composite file with an extent from `[0, )`. A second component is added with an extent `[0, 0)` on the target OST(s). The file data can incrementally be copied to the target replica component. As data is copied to the new replica component the new component's extent end is increased to cover the just-copied range of the file, for example `[0, 1GB)`. The old component's extent start is increased to cover a smaller range at the end of the file, for example `[1GB, )`, and its data is punched by the same amount (or the source objects are simply deleted at the end).

3.9 Unaligned components

For applications that have poorly-structured IO, it is possible that the application writes a short file header, and then reads or writes large-but-unaligned chunks of the file with large requests. For example, if there is a 64kB header, and then a series of 1MB chunks read/written with a 64kB offset from the start of the file. This produces poorly-formed IO to the underlying OST RAID LUNs because it is not aligned with the RAID chunks. It would be possible to specify a compound layout with a Data On MDT component for the start of the file, and then a RAID-0 OST component for the rest of the file. Unlike the components specified in other examples, the unaligned IO component would be flagged to be starting at the end of the first extent, rather than overlapping the first extent.

3.10 Algorithmic Layouts

Use cases for RAID-1/10 are similar to those for replication, except that the former formats are simpler and may be expressed more concisely. For example the RAID-1 form of a layout designating 4 mirrored objects (RAID-1) is small enough to fit in the extra space of a 512 byte MDT inode. This is not true of the analogous composite layout with 4 entries.

RAID-5 and 6 offer increased data availability in the face of OST failures but not at the expense of full mirroring. While we do not anticipate supporting networked RAID-5/6 in the short term, there are interesting use cases for read mostly files. Given an existing (non RAID-5/6) file, a volatile file is opened and given a RAID-5/6 layout, data is copied from the old file to the new, and parity chunks are written. Then the volatile file is merged in to the old file or layout swap is performed.

3.11 Compact Layouts

A widely striped file is to be created in order to achieve maximum IO bandwidth. Transferring a conventional layout for this file requires clients to register tens or hundreds of KB in buffers. Using a compact layout format the file's layout may be transferred with negligible RPC overhead.

4 Solution Proposal

4.1 Layouts for File-Level Replication and Layout Extents

Layouts for file-level replication and extent based layouts will be offered through the same underlying layout type, which we call a *composite layout*. This layout consists of a header (described by `struct lov_comp_md_v1` below), an array of component descriptors (described by `struct lov_comp_md_entry_v1`), and the component layouts (a sequence of blobs that are independent RAID-0/1/5/6/10 layouts of type `struct lov_mds_md_v3` or other layout types in the future).

> This design does not support nested composite layouts (i.e. components which are themselves composites) to avoid complexity and recursion in the implementation of layout handling. It is thought that non-nested layouts provide sufficient flexibility for current projects and anticipated future uses.

```
struct lu_extent {
 __u64 e_begin;
 __u64 e_end;
};

struct lov_comp_md_entry_v1 {
 __u32 lcme_id;      /* unique identifier of component within composite layout */
 __u32 lcme_flags;     /* LCME_FL_STALE, LCME_FL_OUTOFDATE */
 struct lu_extent lcme_extent;  /* file extent for component */
 __u32 lcme_offset;     /* offset of component layout from start of composite layout
*/
 __u32 lcme_size;     /* size of component layout data in bytes */
 union {
  __u64 lcme_padding;
 } u;
};

struct lov_comp_md_v1 {
 __u32 lcm_magic;     /* LCM_MAGIC_V1 */
 __u32 lcm_flags;
 __u32 lcm_size;
 __u16 lcm_entry_count;
 __u16 lcm_layout_gen;
 union {
  __u64 lcm_padding[2];
 } u;
    struct lov_comp_md_entry_v1 lcm_entries[];
};
```

Each component has an extent which describes the range of the file to which the layout applies. The extent does not necessarily need to cover the full file range. Any extents which are overlapping other extents are replicated, and are expected to contain the same file data at the same logical offset. A replicated (overlapping) extent may be marked **UPDATING** if it is currently being updated asynchronously by a client (see File-Level Replication Solution Architecture for more details). A replicated extent may be marked **STALE** if there was a hard failure updating the data of that extent to match the primary replica.

The mechanism for maintaining and resynchronizing replicas is beyond the scope of this document. However, it should be mentioned that it is desirable to keep **STALE** replicas attached to a file rather than removing them immediately upon OST failure. The number of updates needed to resynchronize a **STALE** replica may be minimal if it is offline for only a short time. This may also allow recovery of an old version of the file from a **STALE** replica if the primary replica suffers a fatal error.

> In the design we normally assume that the component extents in a composite layout have the same starting offset at byte 0 of the file. The extents may each form a non-overlapping a subset of [0,), or the may all start at file offset 0, or there may be some other overlap. However, we should try not to use the component extent start as an offset when accessing the component objects. That is, if a component has a single object *O* and extent [*s*,) then the file byte at position *p* should be found at offset *p* of *O* and not at *p* - *s*. In this way an extent with non-zero start can be converted to one which starts at 0. Similarly assume that file data is safely mirrored to another component, a component whose extent starts at zero can be figuratively *punched* to have some positive start without remapping the objects, followed by a punch of corresponding objects data.

The ability to pack independent file layouts as components of a larger composite layout provides a great deal of flexibility, while isolating the complexity of the individual layouts. By allowing both overlapping and non-overlapping extents to be specified, it is possible to construct file layouts for almost any use case. The ability to add different component layouts in the future (e.g. CRUSH, RAID-5/6) will allow flexibility without requiring changes in the core layout infrastructure.

For quota accounting of files with compound layouts, each component is treated in the same way as a separate file with the same contents. Adding a replica to a file will increase the quota usage of a user, and removing a replica will decrease the quota usage. For files with Data-on-MDT, the space usage of the component on the MDT will be accounted separately from the space on the OSTs. With pool-based quotas (a separate project that DDN is currently working on in LU-4017) it would be possible in the future to separately administer the quota space

available to users on the MDT. This allows and encourages users to pick the availability and performance characteristics that suit their needs best.

It is anticipated that replication requirements can also be managed by an external policy engine such as RobinHood, to add or remove replicas to files, to migrate small files to the MDT, or to create or migrate replicas to different tiers of storage.

Potentially complex applications are possible in the future with integration into userspace libraries/applications, such as tailoring file IO characteristics differently for disjoint parts of a single large file. For example, an HDF5 file could use high IOPS OST pool for components with extents covering an index or small IOs, replicated (overlapping) extents for important metadata, and large streaming components for extents with well-formed IO.

4.2 Operations on Composite Layouts

Several kinds of operations are needed to manipulate simple and compound file from a file with a simple layout.

1. A file with a simple layout is converted to a composite layout whose sole component is the previous layout.
2. A file with simple RAID-0 layout is merged into an existing compound file.
3. A replica (composite layout component) of a file is split out to a new file with only this component as its layout.
4. A replica (composite layout component) of a file is split out of the compound layout and is destroyed.
5. A replicated file is destroyed, each layout component is destroyed.
6. From a composite layout, the component with a given id is opened.
7. From a composite layout, the component with a given id set/clears the stale/offline state.
8. A composite layout is repacked after removal of a component.

4.3 Algorithmic Layouts

1. A new RAID-1/5/6/10 file is created.
2. Two suitably striped RAID-0 files are merged into a RAID-1/10 file.
3. A suitably striped RAID-0 file is merged into an exiting RAID-1/10 file.
4. A specified mirror in a RAID-1/10 file is split off into a new file without an assigned layout.
5. A volatile (open unlinked) file with RAID-5/6 layout is created and written with a copy of an existing file's data. The volatile file's layout is merged as a component of the original file. The RAID-5/6 component can be read.

4.4 Compact Layouts

1. A file is to be striped over a large number of OSTs, say 512 or more. An ordinary RAID0 layout for the file would be tens or hundreds of KB in size. Instead of explicitly specifying the FID of each OST object, a bitmap of OST indices is stored along with enough data that for each OST index set in the bitmap, a corresponding OST object FID may be computed.
2. Compact layouts must also include a starting index (or bias) to ensure balanced OST use.

5 Unit/Integration Test Plan

5.1 Composite Layouts

1. Create, store, load, and destroy empty composite layout on a file with no assigned layout.
2. Convert simple file layout to singleton composite layout.
3. Convert singleton composite layout to simple layout.
4. Merge simple file layout to existing composite layout.
5. Split component of composite layout to existing file without layout.
6. Move component between the composite layouts of existing files.
7. Get component layout with a given id from composite layout and validate.

6 Acceptance Criteria

Passes Unit/Integration Test Plan.

*Other names and brands may be the property of others.