File-Level Replication Solution Architecture

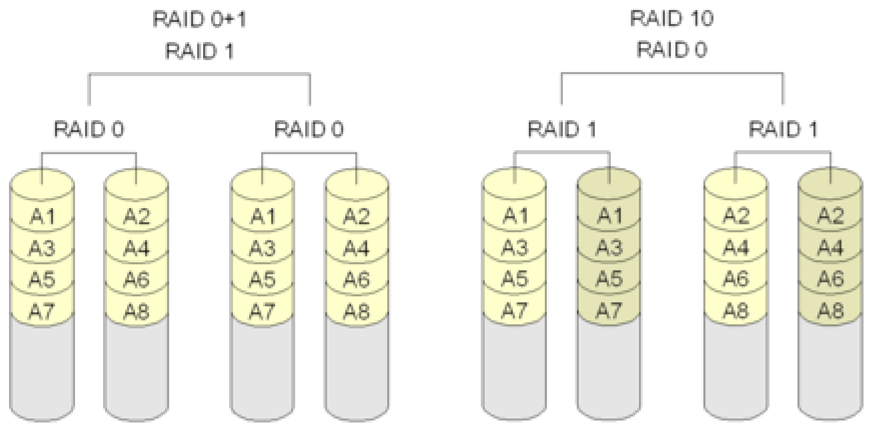
Introduction

The Lustre* file system was initially designed and implemented for HPC use. It has been working well on high-end storage with internal redundancy and fault-tolerant . However, no matter how good storage subsystems are, it still can fail. Lustre software lacks a mechanism to deal with this problem. Some files will become inaccessible if a server is out of service. This problem becomes severe after cloud computing emerges because where usually commodity hardware are deployed which means higher failure rate.

In this document, a solution of replication will be proposed for the Lustre software. With replication, a user can store the same data on multiple OSTs as replicas so that we can tolerate storage failure. IO availability can be improved because we can choose replica to serve an IO. As well, for files that are concurrently read by many clients (e.g. input decks or executables) the replication feature can be used to improve the aggregate parallel read performance of a single file.

However, a complete solution of replication is difficult, so a phase by phase implementation will introduce useful functionality incrementally until the full feature is complete. In the first phase, we're going to work out a technical preview with limited write supported; writing to a replicated file will make it degenerated as a regular file, which means only one replica will be chosen the serve the write and other replicas will be simply marked as stale. An administrator can subsequently return the file into replicated state again with command line tools by synchronizing replicas. There is typically some delay recreating a replicated state following a write, which is know as 'delayed write.'

The Lustre software already supports striping, a.k.a RAID-0. Now we're going to support replication. We can build up complex file layout: striping first and then replicating as RAID-0+1, or replicating first and then striping as RAID-1+0, as follows:



For practical reasons, file replication will initially be implemented as RAID-0+1 in this initial phase so that replicas can easily be created in userspace, and can be heterogeneous across different storage types.

In the rest of this document, in order to describe the file layout easier, we're going to use mirroring interchangeably as replication. The individual replicas of RAID-0+1 could be single OST objects, or RAID-0 striping of OST objects. Anyway, they are mirrored files and won't be distinguished in this document.

Solution Requirements for Read-Only Mirroring

RAID-0+1 and simple RAID-1 support

For simple RAID-1 support, multiple OST objects will store the same data. For RAID-0+1 support, OST objects will be striped and then replicated on top of striped objects.

Mirrored read

This is the core feature of replication. When a process tries to read a mirrored file, and if the first replica is unreachable, the IO engine is able to try the next replica and so on until the read is finished. This way we can improve the IO availability. Mirrored read is also useful to improve the read bandwidth of a read-only file that is accessed by a large number of clients concurrently (e.g. executables, libraries, or input files).

Write to mirrored files

In the first phase of replication project, only delayed write is supported. Write to mirrored files will first mark replicas synchronously **out-of-date**. Only one replica is chosen to serve the write and the other replicas will need to be re-synched by an external utility before they are redundant.

Create mirrored files

Supports to create mirror files through command line tool. Since write is not supported, the only way to create a non-zero-sized mirror file is to merge two or more identical files into a mirrored file, in a manner similar to `lfs swap_layouts`. To prevent modification to the OST objects during the replication step, the objects will be marked immutable. The OSTs will deny any writes and truncates to the object.

Split mirrored files

Support to take a specific replica out of a mirror files by replica index, also support to split a mirrored file into its component replicas. Optionally, the split replicas can be deleted instead of being linked to a new file in the namespace.

Use Cases

Read from a mirrored file

Unlike what we are currently doing for files, to read a mirrored file, we have choices to select which replica to read. The application should be returned with correct data all the time.

Read from a mirrored file with a faulty replica

Assuming that client has already chosen one object from a replica to read. However, that object can become faulty immediately after the selection is complete. The IO engine should detect this case and try another replica in a limited time.

Write to a mirrored file

In the first phase of replication project, write to a replicated file will degenerate the replicated file. Only one replica is chosen as primary to update and others will be marked as **out-of-date**. Afterwards, the read and write requests of this file must go the valid primary replica.

Modification of a mirrored file

Some limited modifications of replicated files are allowed without affecting the content of the file. This includes link, rename, setattr (chown, chmod, utimes, etc), unlink, swap layout.

Solution Proposal

Mirror file layout support

In the past, Lustre software doesn't provide a solution to modify the layout of a file after it has been created. To create and split a replicated layout, we need a complete solution to manage the layout. Different layout may have different ways to support layout change, so that the generic framework of layout operation is provided by another project of layout enhancement. RAID-1 will implement RAID-1 specific layout operations based on that.

Since the initial implementation of replication will not support writing to a mirrored file, the only way to create a mirrored file is to merge two or more identical files, each one will become a replica of the new mirrored file. In this phase the caller must ensure the individual files have exactly the same content before they are merged.

As a temporary solution to allow modifying a mirrored file, a replicated file will be split firstly and then individual files can be modified respectively. Then those files can be merged into a mirrored file again. Notice that in this phase the caller holds the responsibility of making sure that individual files are identical.

Policy to select replica

Policy to select a 'good' replica object to read could potentially be complex. In the ideal case we should select a replica object based on the following information:

- **Status of replica object: If OST object is already in recovery, unreachable, or missing (-ENOENT) definitely we don't want to choose it;**
- Type of replica: prefer to read data from faster OSTs;
- Network topology: read from a closer replica, faster it will be finished;
- Load balance: avoid busy servers.

However, detecting and collecting those information is difficult. It needs support from target, server and network module. The policy will be designed to adapt to such changes in the future.

At present, the only thing we can do is to avoid replica that is already in recovery, and then choose a replica to read by random.

IO framework for read

IO framework is on the client side. Clients must recognize the RAID-1 layout and choose a replica to serve read requests by policies.

The most important work of IO framework is the retry mechanism. For RAID-1 file, the IO RPC should be issued with `rq_no_delay` bit set so that if the object becomes unreachable, the RPC won't be put in the resending list at the RPC layer, but instead it will return to upper layer with an error code. The IO framework should invoke the replica object selection policy to try the next replica object.

The major difficulty of retry mechanism is that this all happens at the PTLRPC daemon context so that operations are restricted to prevent PTLRPC thread from being blocked for a long time.

Dynamic pages will be used to support replica read. In CLIO, a cl_page is composed of two parts: top and sub page. Top page is managed at LLITE layer which represents the vmpage on file level; while sub page is managed at OSC layer which represents information at object level. For a dynamic page, the sub page of a cl_page is dynamic and can be changed on the fly.

When reading a replica failed, the request will be returned with an error. The LOV will intercept this kind of error and then pick another replica to serve the read. In that case, the sub page of the original request will be replaced with new sub pages from new replica, therefore the request can be issued to new OSC.

Delayed write

To implement delayed write, before a write really happens, a dedicated RPC is sent to the MDT. The MDT will choose a replica and modify the file's layout before replying the client. Client will have to refresh its layout before updating the file contents so that it will write the correct replica.

After a replica is chosen and clients have refreshed the layout, the replicated file is degenerated and the IO framework in current the Lustre file system can handle it without any problems.

There is an optimization we can perform. If the file is opened with O_WRONLY, it's highly possible that the file is going to be modified so we can modify the layout in the same transaction as open RPC.

Immutable OST objects

In phase 1 of the replication project, OST objects of a mirrored file may optionally be marked immutable to prevent them from being modified by clients. OST will return `-EPERM` to `truncate()` and `write()` RPCs on the OST. The objects must have the immutable bit cleared after they are split from the replica or when the file is being deleted.

Interoperability

There is no network RPC protocol change for replication implementation. However, new layout has to defined to describe the mirrored file, and there needs to be coordination in case the layout on the file changes, which may produce some interoperability problem.

- 2.3-2.5 clients: They won't recognize RAID-1 layout, but for read-only opens the server can return a non-replicated layout the client (only the primary replica?) for its use. Should the layout be changed by another client (e.g. open for write), then the layout lock will ensure the client will get a new layout. If a replicated file is opened for write by a non-replicating client, it would immediately and synchronously mark the non-primary replicas as **out-of-sync** and only return the primary replica to the client. It may be desirable to have a tunable setting to only allow older clients to open a replicated file in read-only mode, so that they do not invalidate the replica copies.
- 2.2 and older clients: The LAYOUT lock bit is not supported, so these client cannot safely access replicated files.

Command line tools

A set of command line tools will be provided to administrators.

- Create a new replica for an existing file:

  ```
  lfs layout --pattern mirror [--immutable|-I] [other lfs setstripe options] <source_file>
  ```

  where <`source_file`> is an existing file for which a new replica will be added. The `lfs setstripe` options allow specifying the layout for the new replica, using any existing `setstripe` options such as `--stripe-count`, `--stripe-index`, `--pool`, excluding `--stripe-size` which must be identical across all replicas. If no `setstripe` options are specified, then the default layout will be used for the new replica. Care must be taken to ensure that the OSTs of the replica objects do not overlap with those of <`source_file`>, or availability may suffer.

  The `lfs layout --pattern mirror` command will mark the <`source_file`> objects immutable, internally create a volatile file with the specified or default striping, and file contents will be copied from <`source_file`> to the volatile file (located on the same MDT), and finally the objects will be merged into the layout of <`source_file`>.

  The `--immutable|-I` option allows setting the file immutable (read-only) after the replicas are created. That prevents writes to the file, which would mark the replicas **out-of-date**. It can only be opened read-only. The immutable flag can also be set with the standard `chattr` utility, and must be cleared before the file is unlinked.

- Merge an existing replica onto an existing file:
  `lfs layout --merge <source_file> <donor_file>`

  The `lfs layout --merge` command will mark the <`source_file`> objects immutable, and the objects from <`donor_file`> will be merged into the layout of <`source_file`>, and the now-empty <`donor_file`> will be deleted. Care must be taken to ensure that the OSTs of the replica objects do not overlap with those of <`source_file`>, or availability may suffer.

- Split a replica from a mirrored file:

  ```
  lfs layout --split idx <mirror_file> [target_file]
  ```

  where <`mirror_file`> must is an existing replicated file. If option `--split` is specified, the specified replica `idx` will be separated from the mirrored file. If `idx` is `all` then all of the replicas will be split from the file. The index of later replicas in the file will be reduced by one. If `target_file` is specified, it will be created with the layout of replica `idx`. Otherwise, the replica will be extracted as <`mirror_file`>`#{idx}` as suffix.

- Delete a replica from a mirrored file:

  ```
  lfs layout --delete idx <mirror_file>
  ```

  where <`mirror_file`> must is an existing replicated file. If option `--delete` is specified, the specified replicas will be split from the mirrored file and the objects destroyed. If `--idx` is `all` then all non-primary replicas will be deleted, and the layout will be the same as a non-replicated file.

- Delete a mirrored file can be done using normal tools with the `unlink()` system call, for example:
  `rm <mirror_file>`
- Attributes of the mirrored file will be the same as the original file (size, timestamp, UID/GID, etc), except the block count will be the sum of the block count in the replicas.

Unit/Integration Test Plan

1. Merge existing files into mirrored file.
2. Extend a regular file into mirrored file by specifying directories.

3. Split a specified replica index from a mirrored file.
4. Split all replicas from a mirrored file.
5. Delete a specified replica index from a mirrored file.
6. Delete all replicas from a mirrored file.
7. Verify `read()` from mirrored file works.
8. Verify `rename()`, `chmod()`, `chown()` of mirrored file works.
9. Verify `lfs setstripe` of mirrored file works.
10. Verify `write()` to mirrored file marks replicas **out-of-date**.
11. Verify `truncate()` mirrored marks replicas **out-of-date**.
12. Verify `write()` to immutable file returns an error and does not modify the file.
13. Verify `truncate()` of immutable file returns an error and does not modify the file.
14. Verify `read()` from mirrored file with replica on faulty OSTs works.
15. Verify retry mechanism works for `read()`.
16. Verify `unlink()` of mirrored file works.
17. Verify interoperability with 2.4, 2.1 clients.

Acceptance Criteria

Unit test plan passed.