Lustre* Developer Day 2017

Andreas Dilger

May 30, 2017

Multi-Tiered Storage/File Level Redundancy

Multi-Tiered Storage a requirement for upcoming systems

- Migrate NVM<->SSD<->HDD<->Archive, but allow direct access if needed

File Level Redundancy brings significant value/function to **HPC**

- Can use lower-cost commodity single-port storage
- Availability better than HA failover - no need to wait for server recovery
- More reliable than any single device - no single point of failure

Configure redundancy on a per-file or directory basis, for example:

- Mirror only 1 of 24 hourly checkpoints
- 12+3 erasure code large striped files
- Write to SSD, mirror to HDD

Replica 1	Object *j* (PRIMARY) - SSD OST
Replica 2	Object *k* - HDD OST

Multi-Tiered Storage/File Level Redundancy
Phased Implementation

Phase 0: Composite Layouts from PFL project (2.10)

- Plus OST pool inheritance, MDT pools, Project/Pool Quotas

Phase 1: Delayed read-only mirroring - depends on Phase 0

- Manually replicate and migrate data across multiple tiers

Phase 2: Immediate write replication - depends on Phase 1

Phase 3: Integration with policy engine/copytool - needs Phase 1

- Automated migration between tiers based on admin policy/space

Phase 4: Erasure coding for striped files - depends on Phase 2

- Avoid 2x or 3x overhead of mirroring files

Phase 1: Creating Replicas/Mirrors

Replica created by userspace process

File replication tool leverages existing HSM/migrate functionality

- `lfs` or policy engine for replication by file path, user, size, age, etc.

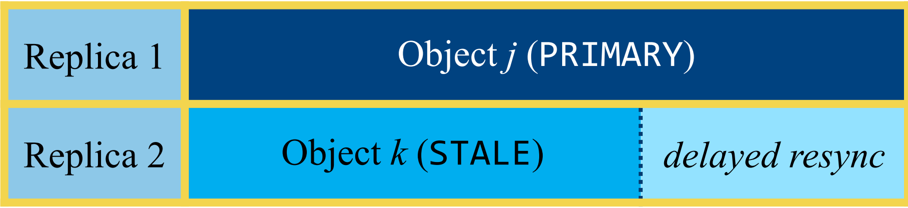
Replica copy attached to existing file, file robust against OST loss

Client kernel does not create replica, can read any replica(s)

- Kernel/user policy to selecting replica

Retry other replica on read timeout

Writes mark other replicas stale



Phase 2: Immediate Write Replication

Client generates write RPCs to 2+ OSTs for each region of the file

- Data page is multi-referenced, do not double RAM, does double IO
- Most files will never have problems, no need for resync in most cases

OST write failure needs sync MDT RPC to mark component `STALE`

- `MDS` generates a ChangeLog record for `STALE` component
- No more writes to that component until it is not `STALE`

Client failure has MDS mark open non-PRIMARY components STALE

`STALE` components resynced from userspace as with PFL Phase 1

Phase 3a: Policy Engine Support

Leverage Policy Engine, copytools to replicate/migrate across tiers

- Functionality starting to appear in RobinHood v3
- Replicate/migrate by policy over tiers (path/file, extension, user, age, size, etc.)
- Release replica from fast storage tier(s) when space is needed/by age/by policy
- Run copytools directly on OSS nodes for fastest IO path
- Partial restore to allow data access before restore or migration completes

Migrate data directly by command-line, API, or scheduler if needed

- Pre-stage input files, de-stage output files immediately at job completion

All storage classes in one namespace = data always directly usable

- Can modify data in place on any tier, no need to migrate files back and forth

Phase 3b: Policy Engine Integration

Improve interaction between Lustre and Policy Engine

Fast inode scan via LFSCK scanner engine

- Scanner can optimize IO ordering better than namespace scanning
- Bulk interface to return many FID+attrs to userspace at once

Internal *OST maps* on MDT to allow fast rebuild of OST?

- Each OST has map index to list each MDT FID with OST FID
- OST map kept uptodate atomically with file creates/object alloc
- On OST failure, file FIDs to resync returned via fast scan interface
- Compare OST rebuild speedup vs. constant overhead of map

Phase 4: Erasure Coded Files

Erasure coding adds redundancy without 2x/3x overhead of mirrors

Add erasure coding to existing striped files *after* write is finished

- Use mirroring for files being actively modified

Suitable for adding redundancy to new/existing striped files

- Add N parity per M data stripes (e.g. 12d+3p)
- Parity declustering avoids IO bottlenecks, CPU overhead of too many parities
  - e.g. split 128-stripe file into 8x (16 data + 3 parity) with 24 parity stripes

dat0	dat1	...	dat15	par0	par1	par2	dat16	dat17	...	dat31	par3	par4	par5	...
0MB	1MB	...	15M	p0.0	q0.0	r0.0	16M	17M	...	31M	p1.0	q1.0	r1.0	...
128	129	...	143	p0.1	q0.1	r0.1	144	145	...	159	p1.1	q1.1	r1.1	...
256	257	...	271	p0.2	q0.2	r0.2	272	273	...	287	p1.2	q1.2	r1.2	...

Phase 4: Erasure Coded File Writes

Hard to keep stripes and parity consistent during overwrite

- Overwrite in place is fairly uncommon for most workloads
- Don't try to keep parity in sync during overwrite
- After FLR Phase 1: mark *parity component* `STALE` during overwrite
  - Resync parity component when overwrite is finished as with mirror
- After FLR Phase 2: create/write temp mirror in addition to parity component
  - Data age determined by allocated blocks in mirror component
  - Merge new writes from mirror into parity when file is idle, skip holes
  - Drop temporary mirror replica after write/merge is finished

Lustre Directory Migration

Layout infrastructure largely developed as part of DNE2

- Migrate single-stripe directory between MDTs

`LMV_HASH_FLAG_MIGRATION` added to allow name hash to be "indeterminate"

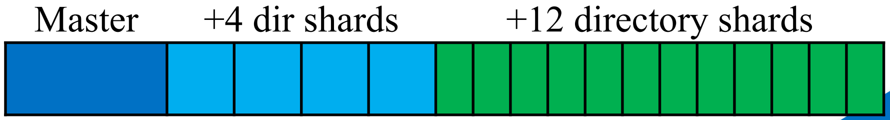
- Client tries MDT for expected hash stripe first, then all other stripes if not found
- Leave a "redirector" on original MDT in case of direct lookup of old FID
- Allows directory migration one entry/inode at a time rather than all at once

Migrate from single-stripe to many-stripe directory

- Add stripes to existing directory, iterate all entries, migrate entires/inodes to new MDT

For directory split, move only entries when directory size hits threshold

- As directory grows larger, remote entries will be a small part of total, FIDs stay the same
- Can migrate entries from M->N stripes



Lustre Metadata Replication

Transaction infrastructure largely developed as part of DNE2

New mirrored directory layout needed, based on striped directory

- Duplicate all directory entries and inodes on multiple MDTs
- Store multiple FIDs in each direntry to locate backup inode with LOV layout

Always replicate top-level dirs to avoid single points of failure

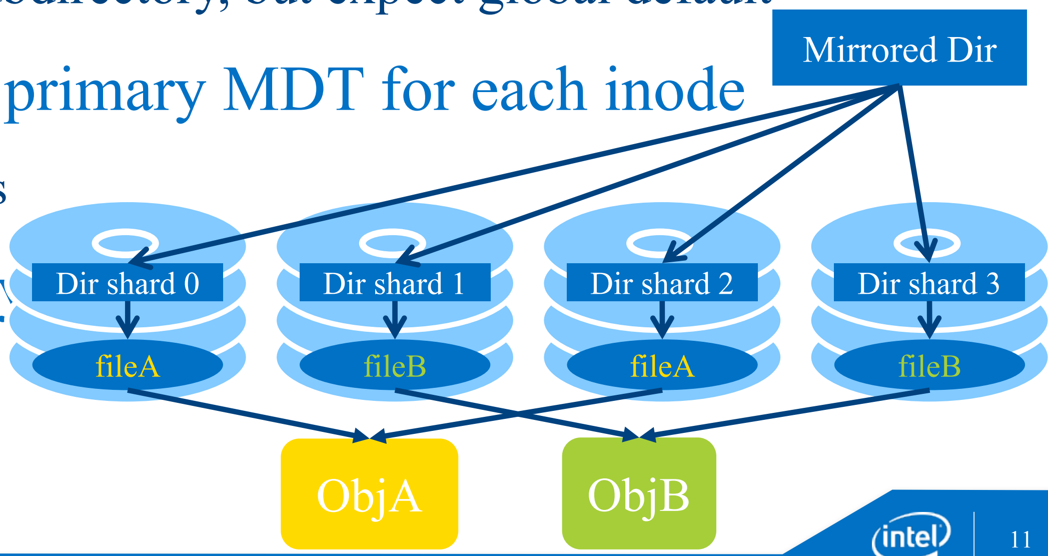
- Replication *could* be tuned per-subdirectory, but expect global default

Modifications are handled by primary MDT for each inode

- Transaction for consistency on MDTs

Client can find backup directly

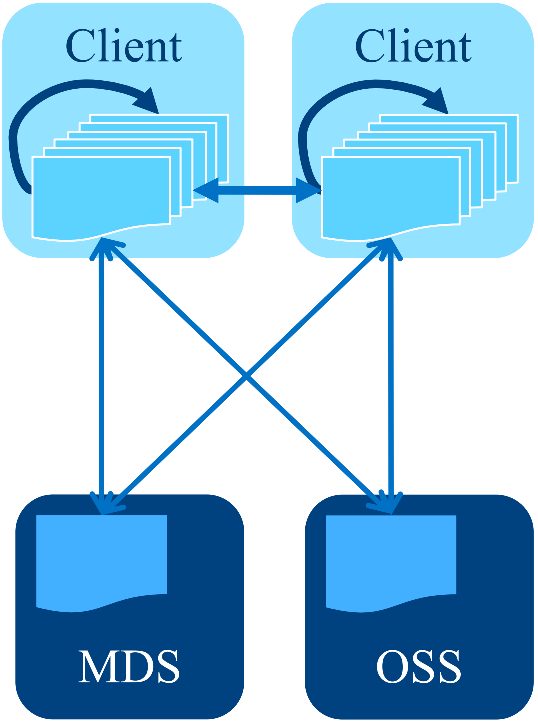
- File layout is mirrored on MDTs
- Data redundancy handled by FLR



Client-side File/Metadata Write Cache

Leverage fast client local NVRAM or RAM cache

- Linux fscache possibly, but still speculative at this point
- Client-side FID generation with directory write lock



Client-internal mount of filesystem image file

- Only one OST object, but a filesystem tree on client
- Low overhead, few Lustre locks, 100k+ IOPS/client?
- Access, migrate, replicate with large read/write to OST
- Some use today via scripts - one client RW/shared RO
- Use with HSM to migrate whole directory tree as file

OSS/MDS/client can export directly for shared use

- Treat as a new temporary MDT for each image?
- Use Data-on-MDT to re-export image to other clients

intel®

FLR Phase 1: Replica File Layout Options

Redundancy based on overlapping composite layouts

- Layout extents overlapping

```
{ lcme_extent_start, lcme_extent_end }
```

  - Each component a *plain* layout (currently RAID-0, but DoM possible in the future)
- Most obvious usage is mirror of single-striped files
- Can have multiple replicas, as many as will fit into a layout xattr
  - 500 single-stripe components about same size as one 2000-stripe RAID-0 layout
- Replicate RAID-0 files, stripe count can be different, stripe size must match
  - For example, if SSD or local OST count doesn't match HDD or remote OST count
- Can also replicate PFL files by having multiple overlapping components

FLR Phase 1: Creating Replicas/Mirrors

Replica initially created by userspace process

- Replica created or sync'd some time after file finishes being written

Any kind of file copy mechanism is OK to use for the replica

- Can be driven directly by user similar to `lfs migrate` or via HSM copytool
- Can use policy engine (e.g. RobinHood) to tune by path, user, size, age, etc.

Replica file copy is composite layout with overlapping extent(s)

- Move copy layout as replica component
- File now robust against OST failure/loss
- Can make replica in different storage tier

Component 1	Object *j*
Component 2	New Object *k*

FLR Phase 1: Read Delayed Replica/Mirror

Client has no idea how replica was created

- Only needs to be able to read the components at this stage

File can be read by any composite-file-aware client

- Normal file lookup gets composite layout describing all replicas
- Read lock any replica OST objects to access data

If Read RPC timeout, retry with other replica of extent

- Read policy can be tuned

Replica 1	Object *j* (PREFERRED)
Replica 2	Object *k*

FLR Phase 1: Select Read Component

Client selects component objects to read based on available extent(s)

- Select component extent(s) that match current read offset, resolve to OST(s)
- Prefer component(s) marked `PREFERRED` by user/policy (e.g. SSD before HDD)
- Skip any OST object(s) which are marked inactive
- Prefer OST(s) by LNet network if OSTs local vs. remote
- If few OSTs left or large file - read same data from each OST to re-use cache
  - Pick components by offset (e.g. component = (offset / 1GB) % num_components)
- If many OSTs left or small file - read data from many OSTs to boost bandwidth
  - Pick components by client NID (e.g. component = (client NID % num_components))

FLR Phase 1: Writing to Read-only Replicas

Write synchronously marks all but one `PRIMARY` replica `STALE` in layout

- Not worse than today when there was never any replica, `STALE` replica is a backup
- Write lock replicas - MDT `LAYOUT` and OST `GROUP EXTENT` all objects to flush cache
- Set `PRIMARY` flag on one replica, `STALE` flag other(s), add `STALE` record to ChangeLog

All writes are done only on the `PRIMARY` component(s) at this point

Resync done after write finished, same way initial replica was created

- Can incrementally resync `STALE` replica
- Clear `STALE` layout flag(s) when done

Can read replicas again as normal

Replica 1	Object *j* (`PRIMARY`)	
Replica 2	Object *k* (`STALE`)	*delayed resync*