Developer Day - FLR & Friends

Andreas Dilger

April 22, 2018

Multi-Tiered Storage/File Level Redundancy

http://wiki.lustre.org/File_Level_Redundancy_Solution_Architecture

Phase 0: Composite Layouts from PFL project (2.10)

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

Phase 1: Delayed resync mirroring (2.11)

- Manually replicate and migrate data across multiple tiers, write invalidates all mirrors

(new) Phase 1.5: FLR infrastructure improvements - needs Phase 1

(new) Phase 2: Erasure coding for striped files (was Phase 4) - needs Phase 1

- Avoid 2x or 3x overhead of mirroring files

Phase 3: Integrate job scheduler/PE/copytool - needs Phase 1

- Automated migration between tiers based on admin policy/space

Phase 4: Immediate write replication (was Phase 2) - depends on Phase 1

- Mirror to two (or more) replicas directly from client - uses 2x write bandwidth

FLR Phase 1.5: Improved Read Replica Selection

Client selects preferred component objects to read - `lov_init_composite()`

- Select component extent(s) that match current read offset
- Prefer component(s) marked **`PREFERRED`** by user/policy (e.g. SSD before HDD)
- Avoid **`STALE`** components or those with any OST object(s) marked inactive
- **(new)** Prefer OST(s) by LNet network if OST is local vs. remote

(new) Few replicas left or large file* - read same data from each OST to re-use cache

- Pick read replica by offset (e.g. `component = (offset / 1GB) % num_replicas`)
- Will distribute shared-file reads across OST replicas evenly by file offset

(new) Many replicas left or small file* - read data from many OSTs to boost bandwidth

- Pick read replica by client NID (e.g. `component = client NID % num_replicas`)
- Will distribute clients evenly between replicas without coordination

* Needs LSOM

FLR Phase 1.5: Improved MDS Object Allocator

MDS *allocates* component objects during open/write - `lod_qos_prep_create()`

- Users/admins can use OST Pools today to segregate replica allocations
- Currently *avoids* allocations from same OST in different components
- **(new)** Avoid OSTs on same NID during replica allocations (no input needed)
  - This will also handle OST failover properly due to shared NID
- **(new)** OST fault-domain awareness to MDS allocator (32-bit domain enough?)
  - OSTs sharing rack/PSU/switch have same domain, no shared domains over replicas

MDS *selects* component during write, marks others stale - `lod_primary_pick()`

- Select components that match current write offset
  - Prefer component(s) marked `PREFERRED` by user/policy (e.g. SSD before HDD)
  - Skip any OST object(s) which are marked inactive
  - **(new)** Prefer OST(s) close to client by LNet network (OSTs local vs. remote)

FLR Phase 1.5: Server-Local Client (SLC)

Allow client to be mounted directly on OSS for more efficient IO

- Intended for HSM copytool and FLR resync operations
- Normal client with LOV/OSC allows access to all file layouts
- Avoid network traffic and round-trip delay for local OST read/write

(new) Mark client `local` on OST (check if local NID)

- Don't add `local` client to `last_rcvd` file to avoid recovery timeout
- Prefer local OST `read()`, remote OST `write()` for copytools
  - Use sync writes initially to avoid cache/memory allocation issues?
- Optimize later to use direct `osc->obdfilter` API calls instead of LNet

Phase 2: Erasure Coded Files (ECF)

ECF adds redundancy with 15-25% overhead vs. 100-200% for mirrors

Add ECF as new component to RAID-0 striped files ***after*** write finished

- Can later use Phase 4 immediate write mirroring for files being actively modified

Suitable for adding redundancy to new/existing RAID-0 striped files

- Add N parity per M data stripes (e.g. 16d+3p) once normal file write is complete (ala RAID-4)
- 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 added parity stripes
  - Could lose *at least* 3 OSTs per file without data loss/interruption, possibly many more

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 2: Erasure Coded File Writes

Hard to keep valid parity during (over)write without overhead

- Overwrite in place **uncommon** for most workloads
  - Most "overwriting" applications write to a new file and rename, for data integrity
- Write data normally, write parity component via resync tool ***after*** data write finished
- Read uses normal RAID-0 OST stripe data unless OST(s) unavailable
- Don't try to keep parity in sync during *overwrite*
- Initially, overwrites mark ***parity*** component `STALE` on MDT
  - Same behavior & infrastructure as Phase 1 Delayed Resync
  - Can only mark parity component `STALE`, can't write file if data OSTs unavailable
- After Phase 4, overwrites create/write temp mirror in addition to parity component
  - Merge new writes from mirror into parity when file is idle, skip holes, drop temp mirror

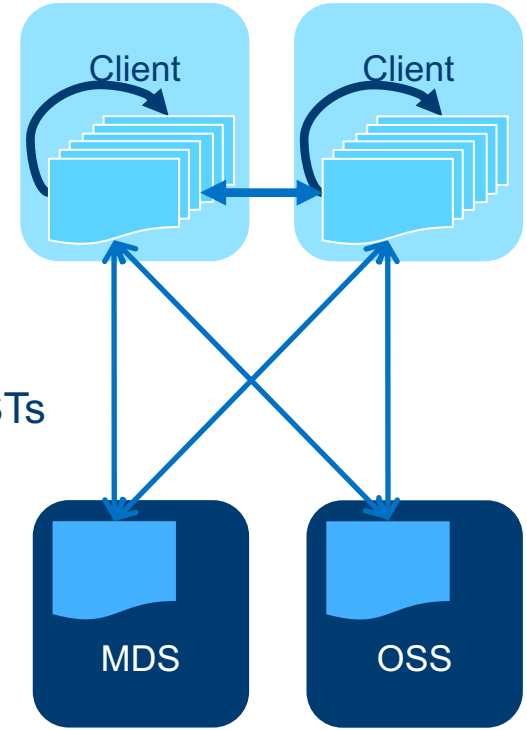
Client-side File/Metadata Write Cache Options

Metadata writeback with client-local RAM/NVRAM cache

- WBC can generate FID on client with directory write lock
- Could open+create+small-write+close with one (delayed) RPC

Client-internal mount of filesystem image file (fileset)

- Image is regular file, but filesystem tree when mounted on client
- Low overhead, few LDLM locks, but 100k+ IOPS/client?
- Access, migrate, replicate with large read/write RPCs to PCC/OSTs
- Some use today via scripts - one client RW or shared RO mount
- Could use with HSM to archive/restore image/tree as one file
- MDS/client could mount and export directly for shared use
  - Data-on-MDT used to re-export image to other clients
  - Client re-export via temporary MDT for each image?



Lustre Directory Migration

Layout infrastructure largely developed as part of DNE2

- Can currently migrate single-stripe directory between MDTs
- **(new)** Add ability to restripe directory, iterate all entries, migrate entries/inodes to new 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
- **(new)** Allow directory migration one entry/inode at a time rather than all at once
- **(new)** Can access directory during migration, except entry being migrated

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

- **(new)** As directory grows, remote entries will be a small part of total, FIDs stay the same
- **(new)** Migrate entries from M->N stripes

Lustre Metadata Replication - Further Out

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 `dirent` to locate backup inode with LOV layout

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

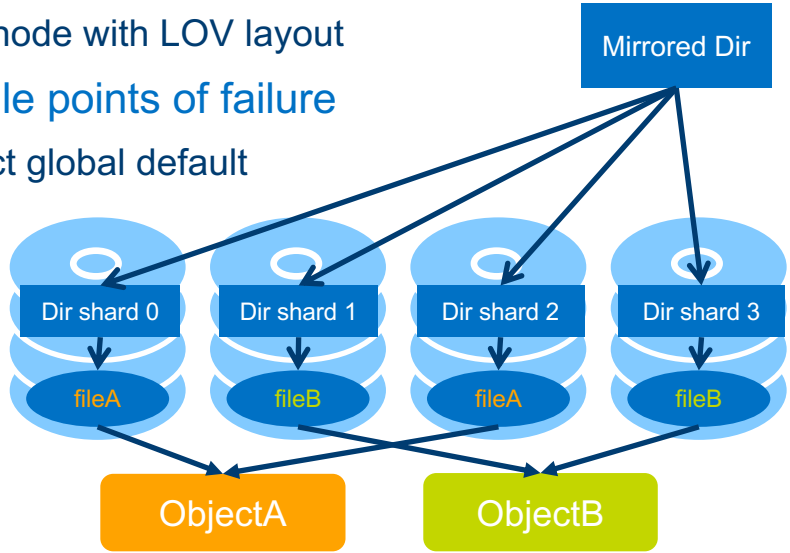
- Replication *could* be tuned per-subdirectory, but expect global default

Changes handled by primary MDT for each inode

- DNE2 transaction for consistency on MDTs

Client can find backup directly

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




intel®
experience
what's inside™

FLR Phase 1: Read Delayed Replica/Mirror (2.11)

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

FLR Phase 1: Writing to Read-only Replicas (2.11)

Write synchronously marks all but one PRIMARY replica STALE in layout

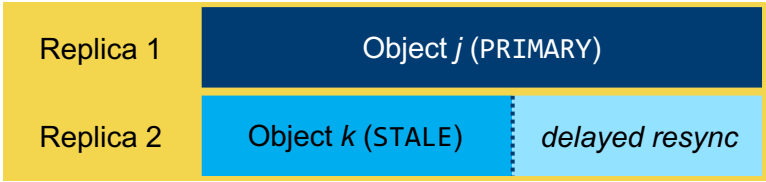
- Not worse than today when there was never any replica, STALE replica is a backup
- Write lock replicas - MDT LAYOUT and OST GROUP EXTENT all objects to flush cache
- Set PRIMARY on one replica, STALE flag other(s), add STALE record to ChangeLog

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



Phase 3a: Job Scheduler/Policy Engine Support

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

- Pre-stage input files for read/write, de-stage output files immediately at job completion
- *Procedural* migration rather than *event-driven* allows earlier usage

Leverage Policy Engine, copytools to replicate/migrate across tiers

- 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 HSM restore to allow data access before restore or migration completes

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

Optimize interaction between Lustre and Policy Engine

Fast inode scan via device-level scanning (Lester/Zester)

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

Internal *MDT Object Index* (MOI) to allow fast rebuild of OST

- MDT has per-OST index to map MDT FID for each OST FID
- OST map updated atomically with file creates/object allocation
- On OST failure, file FIDs to resync returned via fast scan interface
- Compare OST rebuild speedup vs. constant overhead of map

Phase 4: 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