Multi-Tiered Storage and File Level Redundancy

Andreas Dilger

High Performance Data Division

LUG Developer Day April 2016

Improved Data Availability/Flexibility

Software, network, hardware all contribute to Lustre data unavailability

- Lustre at the top of a deep software/hardware stack, depends on all components working
- Needs availability better than individual hardware and software components
- Needs more robustness against data loss/corruption
- Server disk/network bottleneck for files read by many clients (e.g. input files, executables)
- Leverage multiple storage classes dynamically - pre-staged executables and data
- Local vs. remote WAN data access and persistent caches
- Partial HSM file restore for large files - reduce time to first access, access huge data sets
- File versioning to simplify recovery of deleted files

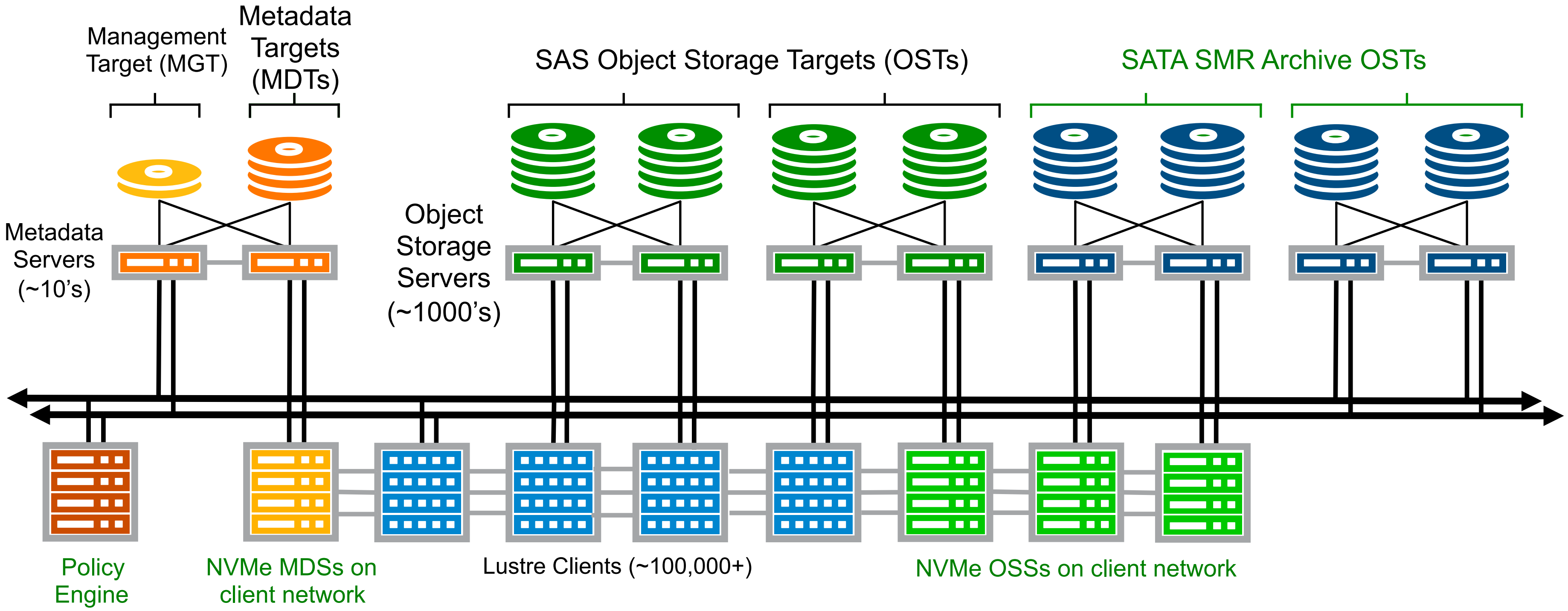
Compound Layouts with File Level Redundancy

Provides significant value and functionality for **HPC** environments

- Availability better than HA failover - no need to wait for failure detection/recovery
- More reliable than any single device - no single point of failure
- Read speed for small shared files - mirror input data across many OSTs
- Replicate/migrate files between storage classes
  - NVRAM<->SSD<->HDD<->Archive but allow direct access from any tier if needed
- Configure redundancy on a per-file or directory basis
  - 2x mirror of one daily checkpoint
  - 128x mirror of read-only input files
  - 12+3 erasure coding of widely-striped files
  - no redundancy on temporary scratch files

Multi-Tiered Storage and File Level Redundancy

Full direct data access from clients to all storage classes



Phased Implementation Approach

Can implement Phase 2/3/4 in any order

Phase 0: Composite Layouts from PFL project

- 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: Integration with policy engine/copytool - with/after Phase 1

- Automated migration between tiers based on admin policy/space

Phase 3: Immediate write replication - depends on Phase 1

Phase 4: Erasure coding for striped files - with/after Phase 1

- Avoid 2x or 3x overhead of mirroring files

Phase 1: Replica File Layout Options

Redundancy based on overlapping composite layouts

- Layout extents with overlapping `{ lcme_extent_start, lcme_extent_end }`
  - Each component a *plain* layout (currently RAID-0, but DoM possible in the future)
- Most obvious 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
- Can replicate striped files, stripe count can be different, stripe size must match
  - For example, if SSD OST stripe count doesn't match HDD OST stripe count
- Can also replicate PFL files by having multiple overlapping components

Phase 1: Creating Replicas/Mirrors

Replica initially created by userspace process

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

Any kind of copy is OK

- Can be driven directly by user via `lfs` similar to `lfs migrate`
- Can use policy engine (RobinHood) policies by path, user, size, age, etc.

Replica copy attached to file as composite layout with overlapping extent(s)

- Simply add layout of copy as component
- File now robust against OST loss

Phase 1: Delayed Read Replication/Mirrors

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

- Access fetches composite layout with replicas
- Read lock any replica to access data

If Read RPC times out, retry with some other replica of that extent

- Policy can be tuned, see next slide ...

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

Phase 1: Selecting Component to Read

Client selects component(s) 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 OSTs(s) which are marked inactive
- Few OSTs left or file is large - read same data from each OST to re-use cache
  - Pick components by offset (e.g. component = (offset / 1GB) % num_components)
- Many OSTs left - read data from many OSTs to increase bandwidth
  - Pick components by client NID (e.g. component = (client NID % num_components))

Phase 1: Writing to Read-only Replicas

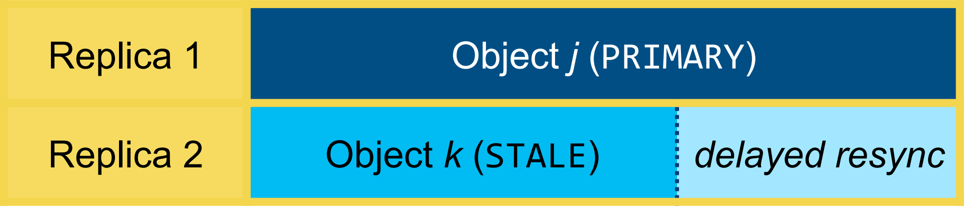
Write synchronously marks all but one `PRIMARY` replica `STALE`

- This is not worse than if there was never any replica
- Write lock all replicas - MDT `LAYOUT` lock and OST `GROUP EXTENT` locks on all objects
- Add `PRIMARY` and `STALE` flags in layout, add `STALE` record into ChangeLog

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

Resync is done after write finished in the same way initial replica was created

- Can do incremental resync
- Clear `STALE` flag(s) from layout



Phase 2: Integration with HSM File Layout

Merge HSM xattr into normal layout as a new file layout type

- Store archive-side file identification into HSM xattr instead of reverse
- Can have multiple archive copies of a single file (e.g. local, offsite)

Restoring part of very large file would have blocked client(s) until restore done

- Chop off end of current component, add a new component after it
- Continue restore in second component (maybe wider striped?), like PFL
- Client can start using first component instead of waiting for whole file

Phase 2: Integration with Policy Engine

Leverage HSM 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 job scheduler if needed

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

All storage classes in one namespace means data always directly usable

Phase 3: Immediate Write Replication

Client generates write RPCs to two or more OSTs for each stripe of the file

- Data page is multi-referenced: does not double memory but does double IO
- Most files will not have any problems, no need for resync in most cases

OST failure during write requires sync RPC to MDT to mark component `STALE`

- `MDS` generates a ChangeLog record for `STALE` component
- No more writes to that component until it is no longer `STALE`

Client failure during write has MDS mark non-PRIMARY components stale

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

Phase 4: Erasure Coded Files

Erasure coding provides redundancy without 2x or 3x overhead of mirrors

Add redundancy component to existing striped files *after* write is finished

- Can add parity component to any existing RAID-0 file

Suitable for striped files - add N parity per M data stripes (e.g. 12d+3p)

- Parity declustering avoids IO bottlenecks, CPU overhead of too many parities
- Should take failure domains into account (avoid data and parity on same OSS)
  - 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 efficiently keep stripes and parity in consistent during overwrite (RAID hole)

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

Legal Information

All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest Intel product specifications and roadmaps

Tests document performance of components on a particular test, in specific systems. Differences in hardware, software, or configuration will affect actual performance. Consult other sources of information to evaluate performance as you consider your purchase. For more complete information about performance and benchmark results, visit http://www.intel.com/performance.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. No computer system can be absolutely secure. Check with your system manufacturer or retailer or learn more at http://www.intel.com/content/www/us/en/software/intel-solutions-for-lustre-software.html.

Intel technologies may require enabled hardware, specific software, or services activation. Check with your system manufacturer or retailer.

You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined". Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

Intel, the Intel logo and Intel® Omni-Path are trademarks of Intel Corporation in the U.S. and/or other countries.

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

intel®

experience what's inside™