Lustre in Finance

A talk mostly about about DNE3, Lustre 2.14/2.15 client/server and some NFS vs Lustre

Steve Crusan and Brock Johnson
Hudson River Trading

Overview

Who is HRT?

2.14/2.15 client/server

DNE3

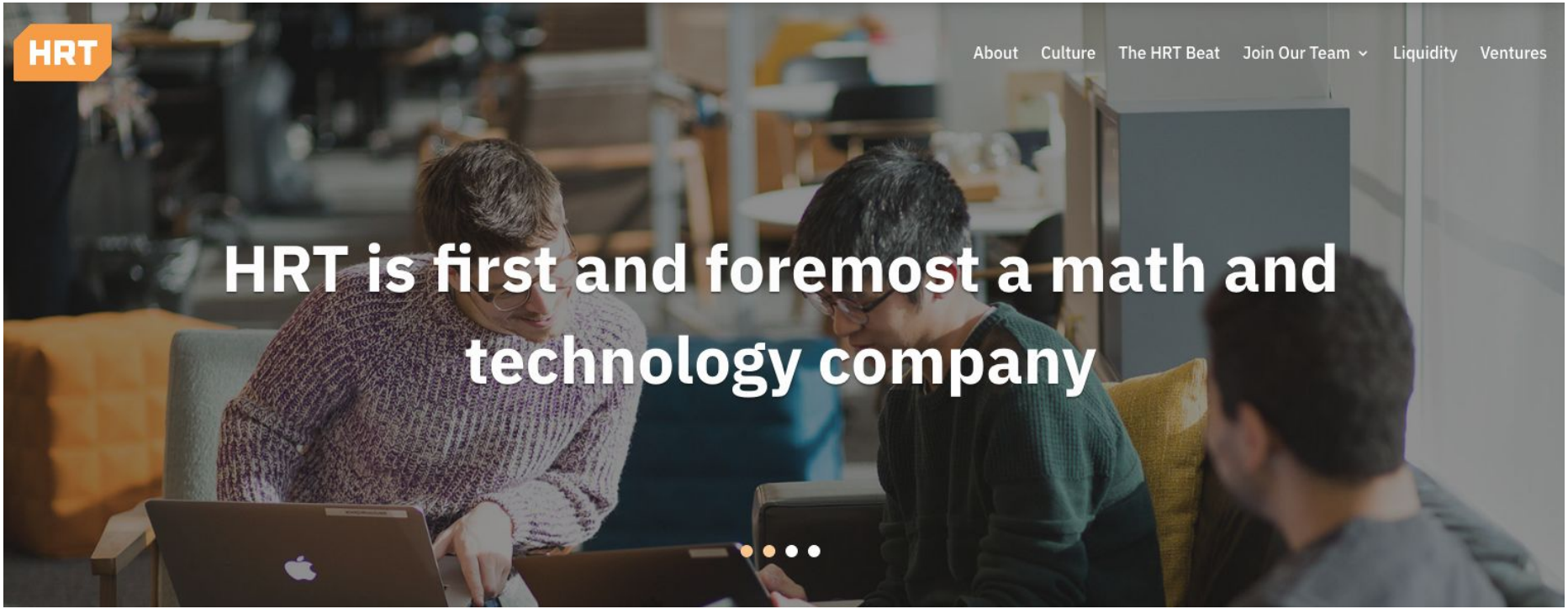
Lustre Metrics and Tooling

Final Thoughts

Questions

Who is HRT?

Hudson River Trading



Hudson River Trading

Hudson River Trading (**HRT**) is a quantitative trading firm headquartered in New York City and founded in 2002. HRT is a committed sponsor to open source software and partnership with the community.

HRT currently has a small, but important Lustre installation.

This talk will cover HRTs experiences onboarding Lustre into our environment, Lustre 2.14 (client/server), DNEv3 (round robin metadata placement), and will showcase some of our tooling that we plan to open source in the future.

Lustre 2.14/2.15

Lustre 2.14/2.15 in production

- I think at this point, we're basically running 2.15 LTS, but regardless, all 99% of clients/servers are at 2.14.0_<some_patch_level>. Whenever I accidentally say 2.14, assume I mean 2.15
- Not our first rodeo, knew there would be some issues running very new client/server code. Someone has to do it, see Stephane Thiell's presentation about 2.12 from LUG19.
- Found a number of issues before we went into production, due to purposely running pathologically horrible workloads
    - Mdtest across many nodes, each creating millions of files/dirs under their own trees. Really wanted to test the DNE3 code under high levels of concurrency. Found a number of bugs.
    - Intermixed IOR/fio w/ the above. Not looking for hero numbers, looking to break things
    - We could've done a better job running the above w/ failovers too. The few times we did do failovers, they seemed to be "okay". As time went on in production, failovers became troublesome, and extremely painful
- Most trouble was from DNE3 MDT-to-MDT (MDT_OUT?) logs and failovers. Even under high loads in production, we didn't really have any failovers. Most failovers caused by bugs in HA mgmt tooling, outside of the core Lustre code.
    - However, when we did need to update our systems w/ newer Lustre code to fix some issues, we needed to failover targets, and that's where the trouble began

Big reason for 2.14: DNE3

- We have plenty of NFS systems, they all have transparent (read mkdir() just works), distributed metadata performance
- DNE1: We are not going to artisan handcraft directory trees, and our codes aren't ever going to be Lustre (or storage system aware, e.g. minus POSIX vs S3), so mkdir() from an application should "do the right thing". DNE1 so only solves this 1 child directory below anyways.
    - Placing each project on an MDT (DNE1) is "okay", but if said "project" needs a ton of aggregate metadata IOPs, then you are limited to the performance of a single MDT.
        - Might not be important for some users, but HRT is very metadata performance sensitive, so not having DNE3 limits the use of the file system to only certain codes. More to life than bandwidth…
- DNE2: Directly from the manpage and presentations, this often isn't recommended, minus maybe at the top of a directory tree, where any child dirs are on a different MDT (same DNE1 issue). If you stripe an entire directory tree from top -> bottom (inherited striping), you need a ton of inodes that point to the remote MDTs, which reduces performance and magnifies the number of inodes required. Only seems useful for singular large directories, with > stripe_count=4 having diminishing performance returns. We tried this pre-2.14, and ran our smaller Lustre system out of inodes/MDT space, and ldlm also went crazy due to the concurrency.
- DNE3: Set a "policy" at the top of a directory tree, and let the client randomly place new child dirs on different MDTs. No application code changes, minimal administrative overhead (well, mostly, :-))

DNE3

DNE3 in practice

- More or less “automatic” (via mkdir()) DNE1
  - If you’ve benchmarked mdtest + optimal/handcrafted DNE1 dirs, expect similar performance
- It’s pretty simple to understand, see man lfs-setdirstripe / lfs-getdirstripe
  - Full example command: `lfs setdirstripe -D -c 1 -i -1 -X <int> -max-inherit-rr <int>`
  - Needs -D (set default layout)
  - -c 1 (1 MDT)
  - -i -1 (any target, similar to OST object creation)
  - -X <depth> (-1 == inherit space balancing forever in a tree)
  - –max-inherit-rr <depth> (try as hard as possible to round robin until below subdir depth X)

Cont. DNE3 in practice

- An example to understand the behavior

```
$ sudo lfs setdirstripe -D -c 1 -i -1 -X -1 --max-inherit-rr 4 lug_2022/
$ $ cat local_create_dirs.sh
#!/usr/bin/env bash
#
#

for i in {0..9}; do
    mkdir sub${i}
    for j in {0..39}; do
        mkdir sub${i}/level_dir_${j}
        sleep 0.1
    done
    cd sub${i}
    sleep 1
done

$ find . -type d -exec lfs getdirstripe -m {} \; | sort -n | uniq -c
     31 0
    317 1
     32 2
     31 3
```

Cont. DNE3 in practice

- That seems unbalanced? No, we went below the –max-inherit-rr depth (4) set on the top level directory depth. After that, Lustre will try to space balance (this test system does have some space imbalance)

```
$ find . -type d -exec lfs getdirstripe -m {} \; | sort -n | uniq -c
     31 0
    317 1
     32 2
     31 3
$ lfs getdirstripe -m sub0/* | sort -n | uniq -c
     10 0
     10 1
     10 2
     11 3
$ lfs getdirstripe -m sub0/sub1/* | sort -n | uniq -c
     11 0
     10 1
     10 2
     10 3
$ lfs getdirstripe -m sub0/sub1/sub2/* | sort -n | uniq -c
     10 0
     11 1
     10 2
     10 3
$ lfs getdirstripe -m sub0/sub1/sub2/sub3/* | sort -n | uniq -c
     41 1
```

Cont. DNE3 in practice

- Notice the behavior once lmv_max_inherit_rr < 1.
  - Once we hit the “sub3” (level 4), no more round robin

```
$ lfs getdirstripe -D sub0/
lmv_stripe_count: 1 lmv_stripe_offset: -1 lmv_hash_type: none lmv_max_inherit: -1 lmv_max_inherit_rr: 3
$ lfs getdirstripe -D sub0/sub1
lmv_stripe_count: 1 lmv_stripe_offset: -1 lmv_hash_type: none lmv_max_inherit: -1 lmv_max_inherit_rr: 2
$ lfs getdirstripe -D sub0/sub1/sub2/
lmv_stripe_count: 1 lmv_stripe_offset: -1 lmv_hash_type: none lmv_max_inherit: -1 lmv_max_inherit_rr: 1
$ lfs getdirstripe -D sub0/sub1/sub2/sub3/
lmv_stripe_count: 1 lmv_stripe_offset: -1 lmv_hash_type: none lmv_max_inherit: -1 lmv_max_inherit_rr: 0
$ lfs getdirstripe -D sub0/sub1/sub2/sub3/sub4/
lmv_stripe_count: 1 lmv_stripe_offset: -1 lmv_hash_type: none lmv_max_inherit: -1 lmv_max_inherit_rr: 0
```

Cont. DNE3 in practice

- Space imbalance. Client code will try to correct the imbalance once we are below the RR depth (normally)

```
$ lfs df -i . | grep MDT
testfs2-MDT0000_UUID   107380736     1987209   105393527   2% /mnt/lustre[MDT:0]
testfs2-MDT0001_UUID   107380736      990442   106390294   1% /mnt/lustre[MDT:1]
testfs2-MDT0002_UUID   107380736      950473   106430263   1% /mnt/lustre[MDT:2]
testfs2-MDT0003_UUID   107380736      804293   106576443   1% /mnt/lustre[MDT:3]
$ lfs df -h . | grep MDT
testfs2-MDT0000_UUID      148.3G        4.6G      130.9G   4% /mnt/lustre[MDT:0]
testfs2-MDT0001_UUID      148.3G        3.8G      131.7G   3% /mnt/lustre[MDT:1]
testfs2-MDT0002_UUID      148.3G        4.1G      131.4G   4% /mnt/lustre[MDT:2]
testfs2-MDT0003_UUID      148.3G        4.5G      131.1G   4% /mnt/lustre[MDT:3]
```

Cont. DNE3 in practice

- What we do in production
  - New 2.15 file systems (or if you are running 2.15 server side w/ MDT0 mounted) automatically set -X -1 –max-inherit-rr 3 at root of file system
    - Probably a good choice if you have an existing system, let space balancing do its thing over time
    - We started fresh on 2.15
    - If you absolutely want to force RR on a particular dataset, set your directory layout, then, set (client side) lctl set_param lmv.<fsname>*.qos_threshold_rr > 20 or 30. Similar to lod.<fsname>*.qos_threshold_rr. Be careful though
  - For any project top level dirs, and/or top level user scratch dirs:
    - `lfs setdirstripe -D -c 1 -i -1 -X -1 -max-inherit-rr 8 <directory>`
  - Why not use –max-inherit-rr -1 ?
    - Not sure, there's no guidance out there, and scans of our other file systems showed a depth of 8 to be good enough for most circumstances.
    - I also felt that having a "cutoff" at a certain depth would help understand the behavior better. Or if there was a bug in the code, it would be easier to debug.
    - Maybe a bit short sighted, but it's been good for us 99% of the time. Whamcloud should probably provide more guidance on this moving forward.

Cont. DNE3 in practice

- Personally, I'd try to set your –max-inherit-rr deeper than you think it should be. Over time round robin should normalize the placement/balance. If you're performance sensitive, and rely on high metadata IOPs rates on all MDSs, space balancing has no concept of performance, and it's likely to hurt you.
  - Make your MDTs bigger than you think you need so space balancing isn't the most important thing in the world
  - An extra 10TB for metadata (cost spread across all MDTs) is less costly for performance than having that for data storage. If your file system can write at 100GB/s, that's 100 seconds of extra capacity at peak write throughput. Not worth it in my opinion.
  - It will take you a lot longer to deal with full MDTs.
  - It will take your metadata intensive applications longer to run if you don't round robin over all MDTs
- Test this out on your own first, before you throw a production application into a DNE3 directory setup
  - As noted, if you're imbalanced already, might need to set lmv.<fsname>*.qos_threshold_rr to ignore space for now.
  - It's easy enough to run mdtest, or even as simple as the toy mkdir script from previous slides
  - Learn lfs getdirstripe (annoyingly, -D option shows different than w/o -D, use both)
- Make sure you are running 2.15 LTS.
  - There were bugs with cached xattrs where remote clients (different from where lfs setdirstripe was run) wouldn't see the updated metadata layout, and there's a plenty of fixes for things we've hit. Take advantage of our pain, :-)

How is DNE3 working out for us?

- Of our hundreds of millions of files (sigh), very balanced (1-2% difference) across all MDTs.
  - Between the time of creating this presentation and presenting it, we've hit some weird issues w/ MDT balancing (more later)
  - Overall, I think it's been pretty damned good
- Our metadata performance mostly scales w/ the number of MDTs
  - If you have 10 MDSs, and each MDS can do 100K getattr/s, 1M getattr/s is easily obtainable w/ good directory round robin layout
  - There's still some directory hotspots, but it's been very minimal. See appendix for some tools to debug hot directories.
  - Remember, the extent of our DNE3 "tuning" was 3-4 lfs setdirstripe commands at a few locations in the filesystem, that's it
- Recently, we've hit some strange issues in regards to balancing that weren't very intuitive
  - See commit message: https://review.whamcloud.com/#/c/45544/
  - This was a big fail, as the Lustre code decided that even though the directory depth (from where lfs setdirstripe) was set was not below the max-inherit-rr level (so > 0), the stickiness placed 98% of directories for a user workload on the same MDT, and said workload created many millions of files (and about 300 subdirs), and was hellbent on stat()ing each and every one of them
  - On clients: lctl set_param lmv.<fs>*.qos_threshold_rr=20 ←I'm tempted to set it even higher
    - This is counter intuitive. I want space balancing if things get out of whack (not waiting until > 20%), but I don't want the code deciding that a directory is too deep and sticking all/most children on the same MDT. This might make sense very deep into the tree (below –max-inherit-rr <depth>), but even so I'd prefer in the future to always round robin unless the space balancing gets out of whack

Cont. How is DNE3 working out for us?

- This pegged almost all metadata IOPs to a single MDT, and eventually caused the MDS load to get out of control, and cause a failover. The failover also hit a bug (that was extremely rare until we applied new fixes for a different recovery problem :-)). Ended up with a number of MDTs waiting for each other to recover, aka a deadlock. Aborting recovery wasn't guaranteed to work, so we Jurassic Park'd the file system (unmount all targets, remount and deal with all the evictions and madness).
- Will be submitting an RFE to disable the internal heuristics where Lustre decides that the overall depth of a directory (regardless of the administrative policy set) is deep enough to disregard admin setup. I understand the reasoning for not making remote dirs forever, but let the admin decide that. If someone loses performance because mkdir and opens/stats deep into a directory tree are too slow, they get what they deserve.
- This performance "enhancement" / stickiness made performance much worse than the extra latency of remote dir lookups

The Bug List … there's others

LU/Link	Description	Impact	Notes
LU-15761	cannot finish MDS recovery	High - Deadlock on recovery	
LU-15645	gap in recovery llog should not be a fatal error	High - failovers evicted all clients	
LU-15644	failed llog cancel should not generate an error	Low - annoying error messages	
LU-15643	do not loop on OI Scrub on same FID	Medium - until oi scrub was disabled, MDT constantly stopped accepting requests, stuck in a loop	
LU-15205	cfs_hash rehashing can race with hash iteration	High - servers crash under load	Found before we went into production
LU-15070	client does not inherit default directory layout if changed	Medium - setting directory layout on 1 client might not be propagated to users, so no DNE3	Found before we went into production
LU-14954	socklnd: fix link state detection	Medium - under certain circumstances, the socklnd code decides it's time to null route LNET traffic, even if the link is fine	Related to incorrect handling of link state changes (such as adding/removing virtual interfaces) - Found before we went into production

Summary of bugs

- In production, mostly MDT-to-MDT recovery/log issues
  - Our metadata intensive codes put a lot of less tested code in the hot path, that's for sure
- We had no experience with these failover scenarios, so learning experience for us
- Whamcloud identified issues quickly, and had patches ready to go in short order
- These issues seemed severe, and they were, but the overall amount of downtime (read filesystem hung/offline) was actually pretty minimal, so I still have a job

Monitoring/Tooling

Now for the boring stuff …

- HRT is a prometheus shop, so *everything* is prometheus based
  - https://www.hudsonrivertrading.com/scaling-prometheus/
- In-house Lustre prometheus exporter
  - Runs on all Lustre clients/servers
  - Powers all of our Grafana dashboards. Not going to show pictures, everyone has seen some iteration of these by now. Probably looks like what y'all have too.
- We also use AlertManager - all alerts based on prometheus queries
  - What we alert on
    - OST/MDT inode/space imbalance > X
      - We use PFL, so mostly worried about MDT imbalance due to performance. Our MDTs were overprovisioned so that we hopefully don't run out of space easily. Losing (ex: 10TB) of data capacity is much better than filling an MDT.
    - Client import states not IDLE/FULL state (see lctl get_param {osc,mdc}.*.import) for long periods of time
    - Server side targets in recovery for > X seconds
    - Standard OST/MDT inode/space fill rates
    - Blackbox I/O tests runtimes
      - "How long does 'ls' take?"
        - Seems stupid/basic, but its what users notice first, and it's usually an indicator of "something" going on
    - Don't let prometheus run lots of statfs() calls from a grid, unless you cache the results (llite.*.statfs_max_age=30); it's an expensive call, and I personally find little use for this on remote file systems, although NFS handles it better…

Cont. Now for the boring stuff …

- We also run a Lustre REST API service on all servers
  - Written in Go, excellent built-in HTTP server w/ good performance and threading
  - Powers some internal tools that yester-year involved SSH+screen scraping
  - Eventually will be the prometheus exporter backend (for servers), but we needed the exporter first, before we developed the API server.
- Accompanying “lladmin” tool
  - Why didn’t you contribute this to the core Lustre code?
    - NIH syndrome of course, much to the dismay of my coworker
    - I’m absolutely terrible at C, so the development cycle would have been much longer, poor(er) code quality, etc.
  - Basically just uses lctl get_param under the hood since Lustre hasn’t yet unified /proc vs /sys, marshals to JSON (or a struct), and bobs your uncle.

lladmin

- Run on any node that has 1 or more file systems mounted that you care about
  - **Not a replacement for lctl/lfs, rather extra functionality**
  - QoL tooling for people not as Lustre savvy (we have shared responsibility for our storage systems, so not everyone has a PhD in Lustre)
    - I'm sure most here know how to get the IP address a target is being served from, but not everyone remembers that, and some feedback on internal documentation is that it's a bit crazy. On our other storage systems, a simple command shows you what you need.
- Simple example, show all usage on a file system (read the quota accounting). There's a post on the mailing list from 14 years ago about a repquota command. Client side tooling automatically connects to API servers (uses osc/mdc import data and a simple yaml config file):

```
$ lladmin dump quotas -f testfs2
+---------+--------+------------+--------+---------+-------+-------+-------+----------+-------+-------+-------+
|  TYPE   |   ID   | FILESYSTEM | TARGET |  USED   | QUOTA | LIMIT | GRACE |  FILES   | QUOTA | LIMIT | GRACE |
+---------+--------+------------+--------+---------+-------+-------+-------+----------+-------+-------+-------+
|    user |   XXX  |    testfs2 |      - | 3.6 TiB |   0 B |   0 B |     0 |   728663 |     0 |     0 |     0 |
|    user |   XXX  |    testfs2 |      - |     0 B |   0 B |   0 B |     0 |      580 |     0 |     0 |     0 |
|    user |   XXX  |    testfs2 |      - |     0 B |   0 B |   0 B |     0 |       12 |     0 |     0 |     0 |
|    user |   XXX  |    testfs2 |      - | 1.9 TiB |   0 B |   0 B |     0 |  1248839 |     0 |     0 |     0 |
|    user |   XXX  |    testfs2 |      - | 3.3 GiB |   0 B |   0 B |     0 |  5312393 |     0 |     0 |     0 |
|    user |   XXX  |    testfs2 |      - | 2.7 TiB |   0 B |   0 B |     0 | 20331782 |     0 |     0 |     0 |
|   group |   YYY  |    testfs2 |      - | 1.9 TiB |   0 B |   0 B |     0 |  1249355 |     0 |     0 |     0 |
|   group |   YYY  |    testfs2 |      - | 3.3 GiB |   0 B |   0 B |     0 |  5312392 |     0 |     0 |     0 |
|   group |   YYY  |    testfs2 |      - | 6.3 TiB |   0 B |   0 B |     0 | 21060453 |     0 |     0 |     0 |
| project |      0 |    testfs2 |      - | 8.2 TiB |   0 B |   0 B |     0 | 27622205 |     0 |     0 |     0 |
+---------+--------+------------+--------+---------+-------+-------+-------+----------+-------+-------+-------+
```

Cont. lladmin

- Ltop -like realtime tools, works fine w/ 1 sec refresh, connecting to all servers simultaneously
  - Doesn't this already exist via ltop? Yes, ltop is great, however I was looking for a unified codebase to do *many* different things
  - Grafana is great, but sampling doesnt show peaks (especially when under load), and sometimes you miss prometheus scrapes

NAME	TIOPS	CLOSE	CROSSDIR RENAME	GETATTR	GETXATTR	LINK	MKDIR	MKNOD	OPEN	RENAME	RMDIR	SAMEDIR RENAME	SETATTR	SETXATTR	STATFS	SYNC	UNLINK
testfs2-MDT0000	56722.84	23374.65	0.00	15716.44	5547.39	0.00	306.25	5889.55	5888.55	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
testfs2-MDT0001	56148.29	22802.85	0.00	17080.20	5437.37	0.00	306.23	5260.82	5260.82	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
testfs2-MDT0002	55804.43	24993.63	0.00	12191.04	5869.06	0.00	298.29	6226.21	6226.21	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
testfs2-MDT0003	59014.50	26601.82	0.00	13031.52	6228.91	0.00	319.28	6416.48	6416.48	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
total	227690.06	97772.96	0.00	58019.20	23082.73	0.00	1230.05	23793.06	23792.06	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

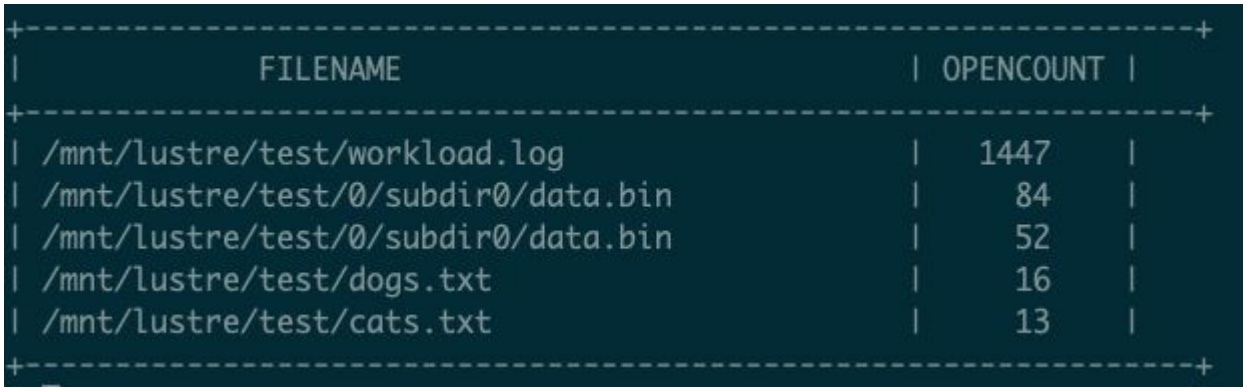
Cont. lladmin

- Data

NAME	TBW	RBW	WBW	TIOPS	RIOPS	WIOPS
testfs2-OST0000	3.5 GB	2.1 GB	1.4 GB	515.2228	248.1441	267.0787
testfs2-OST0001	4.6 GB	2.8 GB	1.8 GB	620.8242	359.7393	261.0850
testfs2-OST0002	2.2 GB	1.6 GB	590 MB	1992.8662	1816.4975	176.3687
testfs2-OST0003	2.5 GB	1.7 GB	784 MB	1628.4132	1482.9124	145.5008
testfs2-OST0004	2.8 GB	1.7 GB	1.2 GB	2658.8719	2507.3351	151.5368
testfs2-OST0005	3.7 GB	2.7 GB	1.0 GB	3999.8721	3746.6399	253.2322
testfs2-OST0006	3.5 GB	1.5 GB	2.0 GB	764.2793	449.4002	314.8791
testfs2-OST0007	2.5 GB	1.9 GB	683 MB	3082.9893	2833.8790	249.1103
-	-	-	-	-	-	-
total	25 GB	16 GB	9.4 GB	15263.3390	13444.5474	1818.7915

Cont. lladmin

- Open files
  - Uses mdt.<target>.exports.<client_nid>.open_files
  - Can use a redis cache so lookups aren't required



FILENAME	OPENCOUNT
/mnt/lustre/test/workload.log	1447
/mnt/lustre/test/0/subdir0/data.bin	84
/mnt/lustre/test/0/subdir0/data.bin	52
/mnt/lustre/test/dogs.txt	16
/mnt/lustre/test/cats.txt	13

Cont. lladmin

- Local client I/O operations
  - This seems like an llstat replacement, but it really just combines a few different equivalent llstat commands. Basically, another QoL for people used to nfsstat.

```
Name: testfs2
Date: 2022-05-09 17:34:37.635033546 -0400 EDT
+------------------+-------+-----------+-------------+----------------+--------------------+
|      METRIC      | UNIT  |    OP/S   |     MIN     |      MAX       | THROUGHPUT/LATENCY |
+------------------+-------+-----------+-------------+----------------+--------------------+
| close            | usec  | 1.97 op/s | 2.00 ms     | 911275.00 ms   | 0.81 avg ms        |
| fsync            | usec  | 2.96 op/s | 42975.00 ms | 13454283.00 ms | 1226.93 avg ms     |
| getattr          | usec  | 1.97 op/s | 1.00 ms     | 362213.00 ms   | 0.00 avg ms        |
| inode_permission | usec  | 49.28 op/s| 0.00 ms     | 4811.00 ms     | 0.00 avg ms        |
| mknod            | usec  | 1.97 op/s | 229.00 ms   | 408179.00 ms   | 0.59 avg ms        |
| open             | usec  | 1.97 op/s | 0.00 ms     | 2358818.00 ms  | 0.01 avg ms        |
| opencount        | reqs  | 1.97 op/s | 1.00 ms     | 139.00 ms      | 0.00 avg ms        |
| truncate         | usec  | 1.97 op/s | 560.00 ms   | 958586.00 ms   | 1.88 avg ms        |
| write            | usec  | 54.21 op/s| 0.00 ms     | 5135143.00 ms  | 15.49 avg ms       |
| write_bytes      | bytes | 54.21 op/s|       20.00 |    16777216.00 | 867.38 MB/s        |
+------------------+-------+-----------+-------------+----------------+--------------------+
```

Cont. lladmin

- Other simple things / toy examples
  - Show all Lustre targets, with their server IP listed (who remembers which host when you have a bunch of targets)
  - Show Lustre target state on a node (including history)
    - Simple to capture via lctl of course, but make it easier
  - Multi-threaded lfs find
    - Spawn lots of threads on a host that take a “root” directory and distribute child dirs amongst other threads
      - Not a dwalk replacement, but lustre aware, and can be distributed amongst many compute nodes with many “subroot” directories

Cont. lladmin

- Very easy to extend the REST API endpoints and add more:
  - Create the core code that parses something from lctl/lnetctl/etc
  - Create methods to marshal to JSON
  - Create an HTTP handler using the above tooling
  - Add unit tests, :-)
  - Compile new binary, place on servers, restart service
- Simple config
  - Servers have REST API keys, credentials, and certificates
  - Client API config needs credentials and pub certificate. Everything else is auto determined via osc/mdc import subsystem data from mounted filesystem(s)
- Client side code that interacts with REST API endpoints on each server is multithreaded (Go routines), and from a previous job I was able to concurrently query 100+ Lustre servers and display realtime stats in less than 1 second, w/ limited CPU/memory usage

This stuff seems interesting, can I use it?

- HRT is cool with me open sourcing the prometheus and Lustre API tooling
  - Need to do all of the paperwork
  - Will post on the lustre mailing list when the code is on github
  - Can also check here periodically: https://github.com/hudson-trading
- Will add more API endpoints as time permits, definitely looking to add realtime jobstats
  - Already have API endpoint, just need to write the lladmin client side code to render

Final Thoughts

Lustre 2.14/2.15, DNE3, Lustre vs NFS, etc

- Shows a lot of promise, and raises bar for Lustre metadata performance!
  - I would personally wait until 2.15.{2,3}, unless you are a cowboy
  - Still some unresolved bugs around directory permissions (if you have 50M directories, 5-10 of them might not have the right permissions). Annoying, but hard to reproduce due to not having time and likelihood of triggering. 0 instances of this w/ the same codes on NFS
  - idle_timeout=20 (default) makes failovers much easier (when they work), since targets don't need to connect to every client and do replays
  - There's bound to be more recovery and bugs as well as race conditions w/ MDT_OUT and OST_OUT communications
  - OSTs are isolated. MDTs are not isolated … at all
    - MDT-to-MDT connections (for remote dirs, etc)
    - MDT-to-OST connections

Cont. Lustre vs NFS, etc

- We haven't thoroughly tested some of the 2.15 data performance improvements, because our data I/O performance was good enough. We are grid computing, not a defensive I/O and/or checkpoint shop, so we don't need 5TB/s of write throughput. However, we achieve better performance (throughput/iops/latency) than our NFS based platforms in most circumstances.
- There are certainly hidden "costs" of Lustre vs NFS
    - Before flash became affordable, Lustre/GPFS could handily beat NFS based systems in throughput. Not totally true anymore, it's a lot closer than it was in the past, and until DNE3, NFS systems w/ some flash for metadata handily beat Lustre in non-data ops.
    - NFS "failover" is stupid simple (move IP address, gratuituous ARP, 10 seconds at most), and the NFS client just works. The same cannot be said for Lustre, whether it be failover scenarios, or even managing the lustre client on many compute nodes. There's an admin burden here that I think most people understand.
- Do the performance improvements outweigh the hidden Lustre costs?
    - For us, it's sometimes…
    - 1 ms vs 2ms for us doesn't matter. No sane trading firm relies on I/O latency for live low latency trading, especially on a network storage system. NFS on flash is pretty good for the average workload, and newer kernels can use multiple TCP connections to a single NFS server, etc.
    - Lustre likely wins in very high end performance scenarios (still a wash on metadata I think, unless there's a ton of MDSs in a file system)
    - Higher throughput/IOPs ceiling is nice, but again we're not a capability/peak shop. Increased I/O parallelism and performance over time might outweigh these costs, but for now, Lustre needs to improve non-performance features a bit more …

How Lustre can improve…

- Disabling the server side page cache when using non-rotational media seems like an anti-pattern
  - I've seen the benchmarks that prove me wrong (faster to disable), and it seems like more of a Linux problem than a Lustre problem, https://jira.whamcloud.com/browse/LU-11347
  - BSD based NFS scale out systems do not have this problem. Can do many GB/s directly out of server side cache w/ almost zero disk IOP/s. In finance, we read a lot of the same datasets from our grid, so this would help. NFS based (linux or BSD) also have in-memory inode caches, which help with performance too.
- Adding/removing targets from a file system is very, very, very cumbersome.
  - It's easy to add storage, but now you have a severe I/O (data or metadata) imbalance problem, especially if you depend on metadata performance being "sharded" across all MDTs. Sitting around and waiting for the problem to fix itself is a non-starter, and asking for trouble. Restriping data and metadata is extremely crazy to do on a many millions of files on a system. NFS appliances and GPFS handle this out of the box.
- DNE3 should be more automatic
  - No other file systems require all of this (why not –max-inherit-rr=-1 always?)
  - I realize some Lustre admins enjoy this level of configuration, so give them the tools
    - It's fairly difficult to keep track of directory depths, and workloads change.

How Lustre can improve...

- The debug tooling is really rough
  - How does the average admin debug a performance problem?
    - Anyone that's good at this should remember your first time
    - Some of the client/server rpc/brw stats and the like are pretty straight forward, but as the problem becomes more complex, now it’s setting debug masks, dumping, parsing
  - Lots of great info in debug dumps, but why no simple tools to help parse and make sense w/o reading the source code and/or google + mailing list + WC Jira?
    - Is the first option to read the source code to decipher?

Cont. How Lustre can improve…

- The last few slides might sounded overly negative/harsh, but regardless our experience has been pretty positive. Very few other scale out systems can do anything close to Lustre, and it's entirely understandable that there will be bumps in the road.
- We put our most challenging workload (that would be a non-starter before DNE3) on Lustre, and it's handled it very well in terms of performance. Stability, etc will improve, especially when we're running an LTS release, :-)
    - We also have a lot of simultaneous "things" running on the system, data syncs, scanning, production I/O, experimental I/O, etc. The combination of everything wouldn't work too well pre-DNE3, or we would be very bottlenecked on metadata operations.
- Lustre gets our seal of approval!



Acknowledgements

Wouldn’t be possible without the hard work of many people at Whamcloud, my coworkers at HRT, and our users!

Questions ?

Thanks for having us!

PS: We're hiring: https://hudson-trading.com/careers/

Appendix - getattr tracing

Run this on an MDS that is showing more getattr operations than others. Can be used to find “hot” directories.

After the debug data is written, grep for ‘mdt_getattr_name_lock’, then post process

Obviously expect to lose at least 10% of overall performance while tracing

```
#!/usr/bin/env bash
#
output_root="/scratch/tmp/getattr_debug"
logfile="${output_root}/lctl_dk_getattr_$(date
+%Y-%m-%d_%H:%M).txt"
sleep_time=120

function reset_masks() {
    echo "resetting subsystem debug mask..."
    lctl set_param subsystem_debug=-1
    echo "resetting debug mask..."
    lctl set_param debug=0
}

mkdir -p "${output_root}"

reset_masks
lctl dk > /dev/null

echo "setting subsystem debug to mds..."
lctl set_param subsystem_debug=mds
echo "setting debug mask to inode..."
lctl set_param debug=inode

echo "sleeping for ${sleep_time} seconds ..."
sleep ${sleep_time}

echo "writing results to ${logfile} ..."
lctl dk > "${logfile}"

reset_masks
```

Appendix - mds_close tracing

Run this on an MDS that is showing more mds_close operations than others. Mds_close is when a client “closes” a directory

After the debug data is written, grep for ‘mdt_mfd_close’, then post process

Obviously expect to lose at least 10% of overall performance while tracing

```
#!/usr/bin/env bash
#
output_root="/scratch/tmp/mdsclose_debug"
logfile="${output_root}/lctl_dk_mdsclose_$(date
+%Y-%m-%d_%H:%M).txt"
sleep_time=120

function reset_masks() {
    echo "resetting subsystem debug mask..."
    lctl set_param subsystem_debug=-1
    echo "resetting debug mask..."
    lctl set_param
debug=ioctl+neterror+warning+error+emerg+ha+config+cons
ole+lfsck debug_mb=128
}

mkdir -p "${output_root}"

reset_masks
lctl dk > /dev/null

echo "setting subsystem debug to mds ..."
lctl set_param subsystem_debug=mds
echo "setting debug mask to +dlmtrace+rpctrace+inode
..."
lctl set_param debug=+dlmtrace+rpctrace+inode
debug_mb=1024

echo "sleeping for ${sleep_time} seconds ..."
sleep ${sleep_time}

echo "writing results to ${logfile} ..."
lctl dk > "${logfile}"

reset_masks
```