Tips and Tricks for Diagnosing Lustre Problems on Cray Systems

Cory Spitz, *Cray Inc. and* **Ann Koehler**, *Cray Inc.*

ABSTRACT: *As a distributed parallel file system, Lustre is prone to many different failure modes. The manner in which Lustre fails can make diagnosis and serviceability difficult. Cray deploys Lustre file systems at extreme scales, which further compounds the difficulties. This paper discusses tips and tricks for diagnosing and correcting Lustre problems for both CLE and esFS installations. It will cover common failure scenarios including node crashes, deadlocks, hardware faults, communication failures, scaling problems, performance issues, and routing problems. Lustre issues specific to Cray Gemini networks are addressed as well.*

KEYWORDS: Lustre, debugging, performance, file systems, esFS

1. Introduction

Due to the distributed nature and large scale of Cray deployed Lustre file systems, administrators may find it difficult to get a handle on operational problems. However, Lustre is a critical system resource and it is important to quickly understand problems as they develop. It is also important to be able to gather the necessary debug information to determine the root cause of problems without lengthy downtime or many attempts to reproduce the problem.

This paper will touch on a broad range of common Lustre issues and failures. It will offer tips on what information to acquire from the Lustre `/proc` interfaces on the clients, routers, and servers to aid diagnosis and problem reporting. Some of the more common and cryptic Lustre error messages will be discussed as well.

Both the traditional Cray Lustre product, which consists of direct-attached systems where Lustre servers are embedded on I/O nodes (SIO or XIO) within the Cray mainframe, and external services File Systems (esFS) offered by the Cray Custom Engineering Data Management Practice (CE DMP) will be covered.

The intended audience for this paper is system administrators, operators, and users that are familiar with Lustre terms and components. Please refer to the Lustre 1.8 Operations Manual (1) included in your CLE release documentation or http://www.lustre.org for definitions. The paper does not address common cases discussed in the Lustre Operations Manual, especially the troubleshooting or debugging chapters.

2. Working with Console logs

One of the first places to look when you suspect a Lustre problem is the console log. On Cray direct-attached systems, most of the console output from Lustre clients and servers are funnelled into the console log on the SMW. Some of the minor `printk`-level Lustre, LNET, and LND messages are recorded in the syslog `messages` file on the SDB node as well. On systems with esFS, the logs could be spread out over the servers or funnelled up to the external service Maintenance Server (esMS), but the client and router messages are still routed up to the mainframe's console log on the SMW.

Lustre log messages can be overwhelming as Lustre is extremely chatty, especially at scale. For instance if an OSS fails, each client and the MDS will become quite vocal as requests fail and are subsequently retried. In systems with thousands of clients, Lustre can easily generate hundreds of thousands of lines of console output for a single problem. `printk` limiting can only go so far and tricks need to be employed by administrators to make sense of it all.

2.1 How to read a console log

The first thing you should do when investigating console logs is to separate the server logs from the clients and from one another. When the Lustre logs are so voluminous that sequential messages from a single host can span pages, they can be very difficult to follow; separating each server's logs makes it much easier to understand what is going on with a given server for a given timeframe. Cray has written a script, `lustrelogs.sh` [Appendix A], that pulls the server identities from the logs and writes out per-server logs. Since the tool does not require the configuration from the Lustre MGS or the `<`*`filesystem`*`>.fs_defs` file, it can be used even if failover occurred.

After the logs are separated, it is much easier to see what is happening and to identify which clients are affected. Those clients can then be inspected separately. Since Lustre is a client-server architecture, understanding the interaction is imperative to determining the root cause of failure.

Lustre identifies endpoints based on their LNET names and you will need to understand this to identify which nodes are specified in the logs. On Cray systems, you can determine what nodes messages refer to by the strings `<#>@ptl` or `<#>@gni`. The number indicated is the nid number and the second half of the name is the LNET network. For example, `10@gni` identifies `nid00010`. Unfortunately, Cray console log messages are prefixed with the Cray cname so it will be beneficial to cross-reference the nid name with the output from `xtprocadmin` or the `/etc/hosts` file. Endpoints on the esFS Infiniband network look like `<`*`IP-address`*`>@o2ib`. The IPoIB address is used as a name even though IP is not used for `o2iblnd` LNET traffic.

Lustre error messages often include strings of cryptic data with an embedded error type or return code, typically `rc`, that clarifies the error once the code is deciphered. These return codes are simply Linux/POSIX *errno* values. Keeping `errno-base.h` and `errno.h` from `/usr/include/asm-generic` handy will make scanning the logs much more efficient. Then when a node complains that, '`the ost_write operation failed with -30`', you will know that it was because the file system was mounted (or re-mounted as) read-only as `-30` equates to `EROFS`.

2.2 What to look for in the console logs

The next step is to identify any major system faults. Look for the strings '`LBUG`', '`ASSERT`', '`oops`', and '`Call Trace`'. Cray systems enable `panic_on_lbug`, so the first three will result in node panics. '`Call Trace`' messages may be the result of watchdog timers being triggered, but typically, if you find any one of these messages, the node most likely hit a software bug. You will need to dump the node and warmboot.

Next, verify that Lustre started correctly. Due to the resilient nature of Lustre, problems that occur while starting the file system may not always be obvious. The console log contains a record of the start process. The section below entitled "Lustre Startup Messages" describes what to look for.

Once you have ruled out a system panic or Lustre not starting, then examine the log for other common failures. These are commonly signalled by the strings '`evict`', '`suspect`', and '`admindown`'. Messages containing these strings may be the result of a bug or some transient condition. You will need to keep looking to further diagnose the issue.

2.2.1 Lustre Startup Messages

Each server issues an informational message when it successfully mounts an MDT or OST. In Lustre 1.8.4, the message is of the form '`Now serving <`*`object`*`> on <`*`device`*`>`'. The string '`Now serving`' uniquely identifies these messages. In earlier Lustre versions, the message is '`Server `*`object`*` on <`*`device`*`> has started`'. Search the log for the string '`Lustre: Server`'. There may not be a message for each OST due to message throttling, but there should be at least one such message from each server. These messages tend to be clustered together in the log. When you find one, examine the surrounding messages for indications of mount failures. For example, if quotas are enabled, the file system can start but fail to enable quotas for OSTs. In this case, you will see the message '`abort quota recovery`'.

Once the devices are mounted, the MDS and OSSs attempt to connect with one another. Connection failure messages are normal at this stage since the servers start at different rates. Eventually, all OSSs will report '`received MDS connection from <#>@<`*`network`*`>`'. If they do not, look for networking errors or signs that a server is down.

If all the servers start successfully, then each client will report mounting Lustre successfully with the message '`Client <`*`fsname`*`>-client has started`'.

You can find more information about events during start up in the output from the `` `/etc/init.d/lustre start` `` or `` `lustre_control.sh <filesystem>.fs_defs start` `` command. A copy of the output is logged to a temporary file on the boot node named `/tmp/lustre_control.`*`id`*. If you suspect a problem with the file system configuration, try running the `` `lustre_control.sh <filesystem>.fs_defs verify_config` `` command for clues to what may be wrong. The `verify_config` option checks that the device paths for targets match what is specified in the

`<filesystem>.fs_defs` file. You may also want to try logging onto the server and issuing the `` `mount -t lustre` `` command directly. Alternatively, you can sometimes gain insight by mounting the device as an ldiskfs file system, just use `` `mount -t ldiskfs` `` instead. In that case mounting the device read-only is preferable. The journal will be replayed upon mount. If there are errors received here, it is an indication that major corruption has occurred and that the device needs repair.

2.2.2 Client Eviction

Client eviction is a common Lustre failure scenario and it can occur for a multitude of reasons. In general, however, a server evicts a client when the client fails to respond to a request in a timely manner or fails to ping within two ping intervals, which is defined as one-quarter of the `obd_timeout` value. For example, a client is evicted if it does not acknowledge a glimpse, completion, or blocking lock callback within the `ldlm_timeout`. On Cray systems, the `ldlm_timeout` defaults to 70 seconds.

A client can be evicted by any or all of the servers. If a client is evicted by all or many servers, there is a good chance that there is something truly wrong with that client. If however, a client is only evicted by a single server, it could be a hint that the problem is not on the client or along its communication path and might instead indicate that there is something wrong with that server. In addition, for these cases, interesting behavior can occur if a client is evicted from an MDS, but not the OSSs as it might be able to write existing open files, but not perform a simple directory listing.

A client does not recognize that it has been evicted until it can successfully reconnect to a server. Since the server is unable to communicate with the client, there is little reason to attempt to inform it since that message would likely fail as well. Eventually the client will notice that it is no longer connected. You will see '`an error occurred while communicating`' along with the error code `-107`, which is `ENOTCONN`. The client will then attempt to reconnect when the next ping or I/O request is generated. Once it reconnects, the server informs the client that it was evicted. Eviction means that all outstanding I/O from a client is lost and un-submitted changes must be discarded from the buffer cache. The string '`evicting client`' will denote when the server drops the client and the string '`evicted`' will pinpoint when the client discovered this fact. The client message is, '`This client was evicted by service; in progress operations using this service will fail`'.

During this time, the client's import state changes to `EVICTED`. Use `` `lctl get_param *.*.state` `` or `` `lctl get_param *.*.import` `` to get detailed information about the history and status for all connections. More information about the import interface is detailed in section 3.2.

Attempted I/O during this window will receive error codes such as `-108`, `ESHUTDOWN`. Most common would be `-5`, `EIO`. Typically, applications do not handle these failures well and exit. In addition, `EIO` means that I/O was lost. Users might portray this as corruption, especially in a distributed environment with shared files. Application writers need to be careful if they do not follow POSIX semantics for syncs and flushes. Otherwise, it is possible that they completed a write to the buffer cache that was not yet committed to stable storage when the client was evicted.

There are twists on this common failure scenario. One of which is when a client is evicted because it is out of memory, the so-called OOM condition. OOM conditions are particularly hard on Cray systems due to the lack of swap space. When the Linux kernel attempts to write out pages to Lustre in order to free up memory, Lustre may need to allocate memory to set up RDMA actions. Under OOM, this can become slow or block completely. This behavior can result in frequent connection and reconnection cycles and not necessarily include evictions. This will generate frequent console messages as the kernel can move in fits and starts on its way out of OOM.

Router failures could be another cause of client evictions. In routed configurations for esFS, both clients and servers will round-robin messages through all available routers. Servers never resend lock callbacks, so clients could be evicted if the router completely fails or drops the callback request. Clients will eventually retry RPC transmissions so they are not as prone to suffer secondary faults after router failure. The point here is to examine all routers between the client and server for failure or errors if clients are unexpectedly evicted.

2.2.3 Watchdog timers

In the case where server threads stall and the Lustre watchdog timer expires, a stack trace of the hung thread is emitted along with the message, '`Service thread pid <pid> was inactive for <seconds>s`'. These timers monitor Lustre thread progress and are not the Linux "soft lockup" timers. Therefore, the time spent inactive is not necessarily blocking other interrupts or threads. The messages generally indicate that the service thread has encountered a condition that caused it to stall. It could be that the thread was starved for resources, deadlocked, or it blocked on an RPC transmission. The bug or condition can be node or system wide so multiple service threads could pop their watchdog timers at around the same time. Cray configures servers with 512 threads, so this can be a chatty affair.

2.2.4 Lost communication

Another class of problem that is easiest to identify from console messages are connection errors such as -107 and -108, ENOTCONN and ESHUTDOWN respectively. Connection errors indicate problems with the Lustre client/server communications. The root cause of connection failures can lie anywhere in the network stack or the Lustre level. No matter where the root problem lies, there will be information from the LNET Network Driver, or LND, since it is the interface to the transport mechanism. On Cray SeaStar systems, the LND is `ptllnd` and on Gemini systems, it is named `gnilnd`. The Infiniband OFED driver, which is used for the external fabric of esFS installations, is called `o2iblnd`. Look for the LND name, i.e., `ptllnd`, `gnilnd`, or `o2iblnd` to help pinpoint the problem. Those lines by themselves may not be that useful so be sure to look at the preceding and following messages for additional context.

For example, on SeaStar systems, you might find '`PTL_NAL_FAILED`' and '`beer`' (Basic End-to-End Reliability) messages, surrounding `ptllnd` messages that would indicate that portals failed underneath the LND.

Alternatively, on a Gemini system, you might find '`No gnilnd traffic received from <nid>`', which could suggest a potentially failed peer. The `gnilnd` keeps persistent connections with keep alive and the local side will close a connection if it does not see receive (rx) or keep alive traffic from the remote side within the `gnilnd timeout`. The default `gnilnd timeout` is 60 seconds. The `gnilnd` will close those connections with the message above when there is no traffic and later re-establish them if necessary. This will result in extra connect cycles in the logs.

If messages with the LND name do not pinpoint the problem, look for a downed node, or SeaStar or Gemini HW errors to explain the lost connection. Generally, in the absence of any of these messages the problem is the higher level Lustre connection. Connection problems can be complicated especially in routed configurations and you may have to collect additional data to diagnose the problem. We will cover data collection for `gnilnd` and routers in future sections.

Very high load and, as discussed earlier, OOM conditions may trigger frequent dropped connections and reconnect cycles. The message containing the strings '`was lost`' and '`Connection restored`' bound the interval.

2.2.5 Node Health Checker

The Cray Node Health Checker (NHC) executes system integrity checks after abnormal application exit. Usually, a Lustre file system check is included. The NHC Lustre test checks that the compute node can both perform metadata and I/O by executing a `statfs()` and creating, opening, unlinking, then writing a single file. That file is created with no explicit file stripe settings and so the test does not necessarily check every OST in the file system.

If the test passes then it will do so silently and you can be assured that most of Lustre is working well. If the test fails, the node will be marked as `suspect` as seen by `xtprocadmin`. Then the test is repeated, by default, every 60 seconds for 35 minutes. Oftentimes if there is a server load issue or transient network problem, then a node can be marked as `suspect` and later pass the test and return to the `up` state.

If the test fails all retry attempts, '`FAILURES: (Admindown) Filesystem_Test`' will appear in the console logs and the node is marked `admindown`. NHC will also stop testing the node. If there is a failure, look at the `LustreError` messages that appear between the time the node is set suspect and the time it is set `admindown` as those messages may offer stronger clues to what has gone wrong.

2.2.6 Cray Failover and Imperative Recovery

Lustre failover on Cray direct-attached systems leverage both Lustre health status and Cray's RCA heartbeat mechanism to determine when to begin failover. (2) Failover for esFS is built around `esfsmon`. (3) For either solution, the backup server reports that it '`will be in recovery for at least <time> or until <#> clients reconnect`'.

If imperative recovery is enabled, which is only available for Cray direct-attached systems, the message '`xtlusfoevntsndr: Sent ec_rca_host_cmd:`' indicates that the imperative directive to clients was sent. '`Executed client switch`' indicates that the client side imperative recovery agent, `xtlusfoclntswtch`, made the import switch. '`Connection switch to nid=<nid> failed`' indicates failure.

2.2.7 Hardware RAID errors

Linux, and in turn Lustre, do not tolerate HW I/O errors well. Therefore, Lustre is sensitive to HW RAID errors. These errors are not included in the console log and may not be included in the `messages.sdb` log either. Typically, errors stay resident on the RAID controller, but SCSI errors will be seen in the console log if, for example, the device reports a fatal error or the SCSI timeout for a command is exceeded. When this happens, the kernel forces the block device to be mounted read-only. At that time, Lustre will encounter the *errno*, `-30`, or `EROFS`, on the next attempt to write to that target. Be sure to match that up with '`end_request: I/O error`' to ensure it was a HW and not a Lustre error that caused the target to be remounted read-only.

2.2.8 Gemini HW errors

Hardware errors reported about the HSN are recorded in special locations such as the `hwerrlog.<timestamp>`, `netwatch.<timestamp>` or `consumer.<timestamp>` event log, but the errors will be evident in the console log as well. These errors are not necessarily fatal, however.

Cray XE systems have the ability to reset the HSN on the fly in order to ride through critical HW errors that would traditionally have resulted in kernel panics, a wedged HSN, or both. The `` `xthwerrlog -c crit -f <file>` `` command will show the critical HW errors. When the Gemini suffers such a critical error, the `gnilnd` must perform a so-called stack reset, as all outstanding transmissions have been lost with the HW reset.

When a stack reset occurs, there will be lots of console activity, but you will see the string '`resetting all resources`'. You will also see the error code `-131`, `ENOTRECOVERABLE`. Remote nodes communicating with the node that underwent the stack reset should receive the error code `-14`, `EFAULT`, indicating that the RDMA action failed. Thus if that error is emitted then the remote peer should be checked for a stack reset condition. Additional `gnilnd` error codes and meanings are explained in Appendix C.

The goal of the stack reset is to keep the node from crashing. The Gemini NIC must be reset to clear the errors, but Lustre can often survive the HW error because the `gnilnd` pauses all transfers and re-establishes connections after the reset completes. However, a stack reset can be tricky and the "old" memory used by the driver for RDMA cannot be reused until it is verifiably safe from remote tampering. The `n_mdd_held` field in the `/proc/kgnilnd/stats` interface shows how many memory descriptors are under "purgatory" hold.

Cray XE systems also have the capability to quiesce the HSN and re-route upon link or Gemini failure. To accommodate this feature, Cray configures both the minimum Adaptive Timeout, `at_min`, and the `ldlm_timeout` to 70 seconds. The long timeouts allow Lustre to "ride through" the re-route, but this is typically an extremely chatty process as many errors are emitted before the system can be automatically repaired.

On the console, the string '`All threads paused!`' will be emitted when the quiesce event completes. Then, '`All threads awake!`' will indicate that operations have resumed. When the quiesce event occurs, the `gnilnd` pushes out all timers so that none will expire during the quiescent period. Moreover, no LND threads are run and new requests are queued and are processed after the LND threads resume. This minimizes the number of failed transmissions.

If an LNET router was affected by either a stack reset or lost a link that was repaired with a quiesce and re-route, then it is more likely that a client could be evicted. This is because although clients can suffer RPC failures and resend, the servers do not resend blocking callbacks.

2.2.9 RPC Debug messages

When you see a message like '`Lustre: 10763:0:(service.c:1393:ptlrpc_server_handle_request()) @@@ Request x1367071000625897 took longer than estimated (888+12s); client may timeout. req@ffff880068217400 x1367071000625897/t133143988007 o101->316a078c-99d7-fda8-5d6a-e357a4eba5a9@NET_0x40000000000c7_UUID:0/0 lens 680/680 e 2 to 0 dl 1303746736 ref 1 fl Complete:/0/0 rc 301/301`' you would gather that the server took an extraordinary amount of time to handle a particular request, but you might throw your hands up at trying to understand the rest. The message is very concise to keep the logs readable, but it is very terse. Messages of this type deserve explanation because they are common and will appear even at the default debug level.

The information in the second half of the message starting at `req@` is pulled from the `ptlrpc_request` structure used for an RPC by the `DEBUG_REQ` macro. There are over two hundred locations in the Lustre source that use this macro.

The data is clearly useful for developers, but what can casual users take from the message? You will quickly learn the pertinent details, but the following explains the entire macro. After the request memory address denoted by `req@` the XID and Transaction Number (*transno*) are printed. These parameters are described in Section 19.2, Metadata Replay, of the Lustre Operations Manual. Next is the *opcode*. You will need to reference the source, but you quickly learn that `o400` is the `obd_ping` request and `o101` is the LDLM enqueue request, as these will turn up often. Next, comes the export or import target UUID and portals request and reply buffers. `lens` refers to the request and reply buffer lengths. `e` refers to the number of early replies sent under adaptive timeouts. `to` refers to timeout and is a logical zero or one depending on whether the request timed out. `dl` is the deadline time. `ref` is reference count. Next `fl` refers to "flags" and will indicate whether the request was resent, interrupted, complete, high priority, etc. Finally, we have the request/reply flags and the request/reply status. The status is typically an *errno*, but higher numbers refer to Lustre specific uses. In the case above, `301` refers to "lock aborted".

The *transno*, *opcode*, and reply status are the most useful entries to parse while examining the logs. They can be found easily because the `DEBUG_REQ` macro uses the eye catcher '`@@@`'. Therefore, whenever you see that in the logs, you will know that the message is of the `DEBUG_REQ` format.

2.2.10 LDLM Debug messages

Another useful error message type is the `LDLM_ERROR` macro message. This macro is used whenever a server evicts a client and so it is quite common. This macro uses the eye catcher '`###`' so it can be easily found as well.

An example client eviction looks like, '`LustreError: 0:0:(ldlm_lockd.c:305:waiting_locks_callback()) ### lock callback timer expired after 603s: evicting client at 415@ptl ns: mds-test-MDT0000_UUID lock: ffff88007018b800/0x6491052209158906 lrc: 3/0,0 mode: CR/CR res: 4348859/3527105419 bits 0x3 rrc: 5 type: IBT flags: 0x4000020 remote: 0x6ca282feb4c7392 expref: 13 pid: 11168 timeout: 4296831002`'. However, the message differs slightly depending upon whether the lock type used was `extent`, `ibits`, or `flock`. The type field will read `EXT`, `IBT`, or `FLK` respectively.

For any lock type, `ns` refers to the namespace, which is essentially the lock domain for the storage target. The two mode fields refer to the granted and requested mode. The types are exclusive mode (`EX`), protective write (`PW`), protective read (`PR`), concurrent write (`CW`), concurrent read (`CR`), or null (`NL`). The `res` field can be particularly handy as it refers to the inode and generation numbers for the resource on the ldiskfs backing store. Finally, for extent locks the extent ranges for the granted and requested area are listed respectively after the lock type. Do not fret what appears to be an extremely large extent size as extent locks are typically granted for a full file, which could support the maximum file size. The typical range is 0->18446744073709551615 which is simply 0xffffffffffffffff or -1.

3. Collecting additional debug data

It is often times necessary to gather additional debug data beyond the logs. There is a wealth of Lustre information spread across servers, routers, and clients that should be extracted. Some of the information is human readable from the `/proc` interface on the specific node. First, we will cover some tools that you can use to gather the data and then we will point out some especially useful interfaces.

3.1 Lustre debug kernel traces

Lustre uses a debug facility commonly referred to as the dk log, short for debug kernel. It is documented in Chapter 24, Lustre Debugging, of the Lustre Operations Manual and the `lctl` man page. However, there are some additional quick tips that are useful, especially when recreating problems to collect data for bug reporting.

Lustre routines use a debug mask to determine whether to make a dk log entry. The default debug mask is a trade off between usefulness and performance. We could choose to log more but then we suffer from reduced performance. When debugging problems it useful to unmask other debug statements in critical sections. To enable all logging, execute `` `lctl set_param debug=-1; lctl set_param subsystem_debug=-1` ``.

The dk log is a ring buffer, which can quickly overflow during heavy logging. Therefore, when enhancing the debug mask you should also grow the buffer to accommodate a larger debug history. The maximum size is roughly 400 MiB, which can be set with `lctl set_param debug_mb=400`.

The current debug mask can be read with `` `lctl get_param debug` `` and can easily be updated using "+/-" notation. For example, to add RPC tracing, simply run `` `lctl set_param debug="+rpctrace"` ``. Desired traces will depend upon the problem, but "rpctrace" and "dlmtrace" are generally the most useful trace flags.

The debug log with a full debug mask will trace entry and exit into many functions, lock information, RPC info, VFS info, and more. The dk log will also include all items inserted into the console log. These details are invaluable to developers and support staff, but because so much information is gathered, it can be difficult to correlate the logs to external events such as the start of a test case. Therefore, the logs are typically cleared with `` `lctl clear` `` when beginning data collection and annotated with `` `lctl mark <annotation>` `` with updates.

The log is dumped with `` `lctl dk <filename>` ``. This method will automatically convert the output format into a human readable format. However, this processing on a busy node may interfere with debug progress. You can also dump the log in a binary format by appending a '1' with `` `lctl dk <filename> 1` ``. This saves a lot of time for large logs on the local node and ensures timely data collection. You can post-process the binary dk log and turn it into a human readable format later with `` `lctl df <binary_dklog> <output_filename>` ``.

Lustre dk logs can be configured to dump upon timeouts or eviction with the tunables `dump_on_timeout` and `dump_on_eviction` respectively. The dk logs can be dumped for other reasons in addition to timeouts and evictions as well. It will be evident in the logs that a dump has occurred because '`LustreError: dumping log to <path>`' will be added to the console log. The path is configurable via `/proc/sys/lnet/debug_path` and defaults to

/tmp/lustre-log. The dumps should be collected if possible. Since the path is well known, there is no reason to first extract the file names from the error messages.

When one of these events occurs, the logs are dumped in binary format and so they will need to be converted with `` `lctl df` `` after they are collected. In addition, the log will contain entries that may be out of order in time. Cray has written sort_lctl.sh included in Appendix B that will reorder the entries chronologically. Another handy utility included in Appendix B is lctl_daytime.sh, which converts the UNIX time to time of day.

3.2 State and stats

In addition to the console and dk logs, there are some special files in the /proc interfaces that can be useful. Snapshots of these files prove useful during investigations or when reproducing a problem for a bug report. The llstat tool can be used to clear stats and display them on an interval.

The client import state /proc interface contains a wealth of data about the client's server connections. There is an 'import' file on the client for each metadata client (mdc) and object storage client (osc). All of the files can be retrieved with `` `lctl get_param *.*.import` ``. The import interface shows connection status and rpc state counts. This file can be monitored to get a quick read on the current connection status and should be gathered when debugging communication problems.

The import file will also include the average wait time for all RPCs and service estimates, although they are brought up to the adaptive timeout minimum (at_min) floor, which again by default on Cray systems is 70 seconds. The timeouts file includes real estimates on network latency. For stats on a per operation basis, inspect `` `lctl get_param *.*.stats` `` to see service counts, min (fastest) service time in µsecs, max (slowest) service time in µsecs, and sum and sum squared statistics.

Failover and recovery status can be acquired with `` `lctl get_param obdfilter.*.recovery_status` `` for an OSS or `` `lctl get_param mds.*.recovery_status` `` for the MDS. It is useful to periodically display the progress with /usr/bin/watch. It is also useful to monitor a select client's import connection as well. This information could be useful if the recovery does not complete successfully.

The nis, peers, and if appropriate with esFS, buffers, routes, and routers files should be gathered from /proc/sys/lnet when investigating LNET and LND problems. They provide the state of the LNET resources and whether peers and routers are up or down. These interfaces are detailed later in section 4.3.

4. Performance

This section will not necessarily tell you how to tune your Lustre file system for performance, but instead it details the causes of performance problems and common sense approaches to finding and mitigating performance problems.

4.1 Metadata performance

One of the biggest complaints about Lustre is slow metadata performance. This complaint is most often voiced as the result of user experiences with interactive usage rather than metadata performance for their applications. Why is that?

Lustre clients are limited to one concurrent modifying metadata operation in flight to the MDS, which is terrible for single client metadata performance. A modifying operation would be an open or create. Although close is not a modifying operation, it is treated as one for recovery reasons. Examples of non-modifying operations are gettatr and lookup.

With enough clients, aggregate metadata rates for a whole file system may be just fine. In fact, across hundreds of clients the metadata performance can scale very nicely in cases like file-per-process style application I/O. But when there are many users on a single node then you've got a problem. This is exactly the situation one finds with the model used on Cray systems with login nodes. Fortunately, any reasonable number of login nodes is supported. Because the nodes cannot use swap, additional login nodes are added to the system as interactive load and memory usage increases. However, if users are dissatisfied with the interactive Lustre performance it would also make sense to add additional login nodes to support more simultaneous modifying metadata operations.

The client's mdc max_rpcs_in_flight parameter can be tuned up to do more non-modifying operations in parallel. The value defaults to 8, which may be fine for compute nodes, but this is insufficient for decent performance on login nodes, which typically use metadata more heavily.

Lustre includes a feature to 'stat'-ahead metadata information when certain access heuristics are met like `` `ls -l` `` or `` `rm -rf` `` similar to how data is read-ahead upon read access. Unfortunately, *statahead* is buggy and Cray has had to disable the feature[1].

In addition to poor single client metadata performance, users often make the problem worse by issuing commands to retrieve information about the file system, which further clogs the MDS and the pipe from each client. Typically, users just want to know if the file system is healthy, but the commands that they issue give

[1] Lustre bug 15962 tracks a deficiency in *statahead.*

them much more information than they might need and thus are more expensive. Instead of `/bin/df` which issues expensive `stat()` or `statfs()` system calls, a simple `` `lfs check servers` ``[2] will report the health of all of the servers. Also, `` `lctl dl` `` (device list) will cheaply (with no RPC transmission) show the Lustre component status and can be used on clients to see whether OSTs are `UP` or `IN` (inactive).

Another way that users can further reduce the metadata load is to stop using `` `ls -l` `` where a simple `` `ls` `` would suffice. Also be advised the `` `ls -color` `` is also expensive and that Cray systems alias `ls` to `` `ls -color=tty` ``. The reason it is expensive is that if file size or mode is needed then the client must generate extra RPCs for the `stat()` or file glimpse operation for each object on an OST. Moreover, the request cannot be batched up into a single RPC so each file listed will generate multiple RPCs (2). This penalty can be very large when files are widely striped. For instance if the file striping is set to '-1', then up to 160 RPCs for that single file will need to be generated (160 is the maximum stripe count.)

Due to the way that the Lustre Distributed Lock Manager (LDLM) handles parallel modifying operations in a single directory, threads can become blocked on a single resource. Moreover, threads must hold a resource until clients acknowledge the operation. Even though Cray systems configure 512 service threads, they can all become quickly consumed due to the blocking. If most of the service threads become serialized then all other metadata services including those for unrelated processes will degrade. This will occur even if there are extra login nodes to spread out the metadata load because the bottleneck is on the server side. Long delays are thus inserted and it can take many minutes to clear out the backlog of requests on large systems. This behavior typically occurs when large file-per-process applications are started that create large numbers of files in a single, shared directory.

There is no good way to identify this condition, but it is useful to inspect metadata service times for file system clients. This can be done quickly by monitoring the mdc import and stats files as described in section 3.2.

4.2 Bulk read/write performance

Lustre provides a variety of ways to measure and monitor bulk read/write performance in real time. In addition, other Linux tools such as `iostat` and `vmstat` are useful, but will not be covered here.

[2] Lustre bug 21665 documents a recent regression with `lfs check servers` that resulted in EPERM errors for non-root users. This regression has been fixed in Lustre 1.8.4 included in CLE 3.1 UP03.

On the client side, `` `lctl get_param osc.*.rpc_stats` `` will show counts for in-flight I/O. `DIRECT_IO` is broken out separately from buffered I/O. This and other useful utilities for monitoring client side I/O are covered in Section 21.2, Lustre I/O Tunables, of the Lustre Operations Manual (1). On the server side, the `obdfilter brw_stats` file contains much useful data and is covered in the same section of the manual.

Use the `brw_stats` data to monitor the disk I/O sizes. Lustre tries very hard to write aligned 1 MiB chunks over the network and through to disk. Typical HW RAID devices work faster that way. Depending on the RAID type, expensive read-modify-write operations or cache mirroring operations may occur when the I/O size or alignment is suboptimal. There are a number of causes to fragmented I/O and `brw_stats` will not indicate why I/O was not optimal, but it will indicate that something needs investigation.

The `brw_stats` file also gives a histogram of I/O completion times. If you are seeing a large percentage of your I/O complete in seconds or even tens of seconds, it is an indication that something is likely wrong beyond heavy load. Oftentimes disk subsystems suffer poor performance without error, or a RAID rebuild is going on. That activity is not visible to Lustre unless the degradation becomes extreme.

The main `brw_stats` file contains all the data for a particular OST. However, per client stats are broken out into `obdfilter.*.exports.*.brw_stats`. This can be used to isolate I/O stats from a particular client.

As part of the Lustre install, the *sd_iostats* patch is applied to the kernel, which provides an interface in `/proc/scsi/sd_iostats/*`. This file can be used to corroborate the `brw_stats`. It is useful because it includes all I/O to the block device, which includes metadata and journal updates for the backing *ldiskfs* file system. As the name implies, the `brw_stats` only track the bulk read and write stats.

Because Lustre performance can degrade over time, it is useful to always keep a watchful eye towards performace. Use `llobdstat` to get a quick read on a particular OST (see Section 21.3.1.2 in the Lustre Operations Manual). The Lustre Monitoring Tool is useful for both real-time (5) and post-mortem performance analysis (6) and is a good best practice to employ. In addition, other best practices such as constant functionality testing and monitoring thereof (7) can be used on a regular basis to spot performance regressions.

The OSS read cache is built on top of the Linux buffer cache and so it follows the same semantics. Thus increased memory pressure will cause the read cache to be discarded. It also works the other way. Increased cache usage can cause other caches to be flushed. Linux does not know what caches are most important and can at times flush much more important file system metadata, such as the ldiskfs buddy maps. Cray Lustre contains an optimization for `O_DIRECT` reads and writes that cause them to always bypass the OSS cache. This extends the POSIX semantics of `O_DIRECT` to the OSS that say, "do not cache this data". However, buffered reads and writes can still exert considerable memory pressure on the OSS so it can be valuable to tune the maximum file size that the OSS can cache. By default the size is unlimited, but it is of little value to cache very large files and we can save the cache and memory space. The read-cache can be tuned by issuing, for example, `` `lctl set_param obdfilter.*.readcache_max_filesize=32M` ``. It can also be set permanently for all OSSs in a file system from the MGS via `` `lctl conf_param <fsname>.obdfilter.readcache_max_filesize=32M` ``.

4.3 LNET performance

LNET performance is critical to overall Lustre performance. LNET uses a credit based implementation to avoid consuming too many HW resources or spending too many resources for communication to a specific host. This is done for fairness. Understanding that credits are a scarce resource will allow for better tuning of the LNET.

Each LND is different, but the `ptllnd`, `gnilnd`, and `o2iblnd` all have a concept of interface credits and peer credits, which is a resource that can only be consumed for a specific peer. There are four kinds of credits relevant to tuning performance: network interface (NI) transmit (tx) credits, peer tx credits, router buffer credits and peer router buffer credits.

The interface credit count is the maximum number of concurrent sends that can occur on an LNET network. The peer credit count is the number of concurrent sends allowed to a single peer. LNET limits concurrent sends to a single peer so that no peer can occupy all of the interface credits.

Sized router buffers exist on the routers to receive transmissions from remote peers and the router buffer credits are a count of the available buffer slots. The peer router buffer credits exist for the same reason that LNET peer tx credits do, so that a single peer cannot monopolize the buffers.

4.3.1 Monitoring LNET credits

The credits are resources like semaphores. Both an interface credit and a peer credit must be acquired (decremented) to send to a remote peer. If either interface or peer credits are unavailable then the operation will be queued.

`/proc/sys/lnet/nis` lists the maximum number of NI tx credits and peer credits along with the current available NI tx credits per interface. When there are insufficient credits, operations queue and the credit count will become negative. The absolute value is the number of tx queued. `/proc/sys/lnet/nis` also records the low water mark for interface credits, which is marked as "min". If this number becomes negative, then more credits may be needed.

LNET and the LNDs also keep track of per peer resources and make them visible in `/proc/sys/lnet/peers`. Most importantly for this view, the two "min" columns track the low water mark for peer router buffer credits and peer tx credits.

These LNET interfaces are documented in Section 21.1.4 in the Lustre Operations Manual.

4.3.2 LNET router performance

The primary consideration for LNET routers is having enough bandwidth to take full advantage of the back-end bandwidth to disk. However, due to the nature of credit based resource allocation, it is possible for LNET routers to choke aggregate bandwidth. For communication to routers, not only must a NI tx credit and peer tx credit be consumed, but a global router buffer and peer router buffer credit are needed.

The LNET kernel module parameters `tiny_router_buffers`, `small_router_buffers`, and `large_router_buffers` account for the global router buffer credits and are visible in the `/proc/sys/lnet/buffers` file. The global router credits really pertain to memory pools of size less than one page, one page, and 1 MiB for the tiny, small, and large buffer tunables respectively. Again, negative numbers in the "min" column indicate that the buffers have been oversubscribed. If the load seems reasonable, you can increase the number of router buffers for a particular size to avoid stalling under the same load in the future.

The number of peer router buffer credits defaults to the LND peer tx max credit count. Therefore, the LNET module parameter `peer_buffer_credits` should be tuned on the routers to allow the global router buffers to be fully consumed.

5. Conclusion

Lustre is a complex distributed file system and as such, it can be quite difficult to diagnose and service. In addition, since Lustre is a critical system resource, it is important to investigate and fix issues quickly to minimize down time. The tips and techniques presented here cover a broad range of knowledge of Cray Lustre systems and are a primer on how to investigate Lustre problems in order to achieve quick diagnosis of issues.

6. References

1. **Oracle.** Lustre Operations Manual S-6540-1815. *CrayDoc.* [Online] March 2011. http://docs.cray.com/books/S-6540-1815.
2. *Automated Lustre Failover on the Cray XT.* **Nicholas Henke, Wally Wang, and Ann Koehler.** Atlanta : Proceedings of the Cray User Group, 2009.
3. **Cray Inc.** *esFS FailOver 2.0.* 2011.
4. **Feiyi Wang, Sarp Oral, Galen Shipman, Oleg Drokin, Tom Wang, Isaac Huang.** Understanding Lustre Filesystem Internals. *http://www.lustre.org/lid.* [Online] 2009. http://wiki.lustre.org/lid/ulfi/complete/ulfi_complete.html#_why_em_ls_em_is_expensive_on_lustre.
5. **Chris Morrone, LLNL.** Lustre Monitoring Tool (LMT) . *Lustre Users Group 2011.* [Online] April 2011. http://www.olcf.ornl.gov/wp-content/events/lug2011/4-13-2011/400-430_Chris_Morrone_LMT_v2.pdf.
6. **Andrew Uselton, NERSC.** The Statistical Properties of Lustre Server-side I/O. *Lustre Users Group 2011.* [Online] April 2011. http://www.olcf.ornl.gov/wp-content/events/lug2011/4-12-2011/1130-1200_Andrew_Uselton_LUG_2011-04-11.pdf.
7. **Nick Cardo, NERSC.** Detecting Hidden File System Problems. [Online] 2011. http://www.olcf.ornl.gov/wp-content/events/lug2011/4-13-2011/230-300_Nick_Cardo.pptx.

7. Acknowledgments

Many thanks to the Cray benchmarking and SPS staff including the field support for always providing the needed data, insights, and operational support in whose experience the authors based this paper on.

Also, thank you to the CUG 2010 attendees for requesting this type of contribution from the Cray Lustre team.

Finally, thank you to Nic Henke for providing insight into `gnilnd` internals. Questions not covered in this paper pertaining to the `gnilnd` internals can be directed to nic@cray.com.

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0001. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency.

8. About the Authors

Cory is the team lead for Lustre integration at Cray. His email address is spitzcor@cray.com. Ann is a file systems engineer at Cray. Ann can be reached at amk@cray.com. Cory and Ann can both be reached at 380 Jackson Street, St. Paul, MN, 55101.

Appendix A

lustrelogs.sh

```
#
# Extract Lustre server messages from console log into separate files
# Copyright 2011 Cray Inc.  All Rights Reserved.
#

#!/bin/bash
# Usage: <script> <console log> [<hosts>]

usage () {
    echo ""
    echo "*** Usage: $(basename $0) [-h] <console_log>"
    echo ""
    echo "*** Extracts MDS and OSS messages from the specified console"
    echo "*** log and places them in separate files based on node id."
    echo ""
    echo "*** File names identify server type, cname, nid, and objects"
    echo "*** on the server. The OST list in file name is not guaranteed"
    echo "*** to be complete but the gaps in the numbers usually makes "
    echo "*** this obvious."
    echo ""
    echo "*** Options:"
    echo "***    -h     Prints this message."
    echo ""
}

while getopts "h" OPTION; do
    case $OPTION in
    h) usage
       exit 0
       ;;
    *) usage
       exit 1
    esac
done
shift $((OPTIND - 1))

if [ "$1" == "" ]; then
    usage
    exit 1
fi
CONSOLE_LOG=$1


# Parses cname and Lustre object from console log messages of the form:
#
# [2010-10-12 04:16:36][c0-0c0s4n3]Lustre: Server garnid15-OST0005 on device /dev/sdb has started
#
# Finds cname/nid pairings for server nodes. Record format is:
# 2010-10-12 04:13:22][c0-0c0s0n3]    HOSTNAME: nid00003
#
# Builds filenames: <oss | mds>.<cname>.<nid>.<target list>
# Extracts records for cname from console file and writes to <filename>

# Lustre Version 1.6.5 and 1.8.2
srch[1]="Lustre: Server"
objfld[1]=4

# Version 1.8.4 and later
srch[2]="Lustre: .*: Now serving"
objfld[2]=3

# Produces: mds:c#-#c#s#n:.MDT0000.MGS or
#           oss#:c#-#c#s#n:.OST####.OST####...
```

```
find_servernodes () {
    local obj_field=$1
    local srch_string=$2

    SERVERS=$( \
        grep "${srch_string}" $CONSOLE_LOG | sort -k ${obj_field} -u | \
        awk -v fld=$obj_field \
            '{match($2, /c[0-9]+-[0-9]+c[0-9]+s[0-9]+n[0-9]+/, cn);
              obj=$(fld)
              sub(/^.*-/, "", obj);
              nodes[cn[0]] = sprintf("%s.%s", nodes[cn[0]], obj);
             }
             END {
                 ndx=0
                 for (cname in nodes) {
                     if (match(nodes[cname], /OST/)) {
                         printf "oss%d:%s:%s ", ndx, cname, nodes[cname];
                         ndx++;
                     }
                     else
                         printf "mds:%s:%s ", cname, nodes[cname];
                 }
            }'
    )
}

# Main

SERVERS=""
for idx in $(seq 1 ${#srch[@]}); do
    find_servernodes ${objfld[$idx]} "${srch[$idx]}"
    if [ "${SERVERS}" != "" ]; then
        break
    fi
done

nid_file="/tmp/"$(mktemp .nidsXXXXX)
grep "HOSTNAME" ${CONSOLE_LOG} > ${nid_file}

echo "Creating files:"
for name in ${SERVERS}; do
    nm=(${name//:/ })
    prefix=${nm[0]};
    cname=${nm[1]};
    objs=${nm[2]};

    nid="."$(grep ${cname} ${nid_file} | awk '{print $4}')

    fname=${prefix}.${cname}${nid}${objs}
    echo "   "$fname
    grep "${cname}" ${CONSOLE_LOG} > ${fname}
done
rm ${nid_file}
```

Appendix B

sort_lctl.sh

```
#!/bin/bash

#
# Sort Lustre dk log into chronological order
# Copyright 2011 Cray Inc.  All Rights Reserved.
#

INF=$*

for inf in $INF; do
        cat $inf | sort -n -s -t: -k4,4 > $inf.sort
done
```

lctl_daytime.sh

```
#!/bin/bash

#
# Convert dk log into time of day format
# Copyright 2011 Cray Inc.  All Rights Reserved.
#

if [ $# -lt 2 ]; then
    echo "usage: $(basename $0) <input_file> <output_file>"
    exit 1
fi

awk -F":" '{ format = "%a %b %e %H:%M:%S %Z %Y"; $4=strftime(format,
$4); print}' $1 > $2
```

Appendix C

gnilnd error codes and meanings from Cray intranet http://iowiki/wiki/GeminiLNDDebug

NOTE: The *text description from errno.h* is provided to reference the string printed from things like strerror and doesn't reflect the exact use in the gnilnd. Some errors are used in a bit of a crafty manner.

Error code (name)
text description from errno.h - description of error(s) in the gnilnd

-2 (-ENOENT)
No such file or directory - could not find peer, often for *lctl --net peer_list, del_peer, disconnect*, etc.

-**3 (-ESRCH)**
No such process - RCA could not resolve NID to to NIC address.

-5 (-EIO)
I/O error - generic error returned to LNET for failed transactions, used in gnilnd for failed IP sockets reads, etc

-7 (-E2BIG)
Argument list too long - too many peers/conns/endpoints

-9 (-EBADF)
Bad file number - could not validate connection request (datagram) header - like -EPROTO, but for different fields that should be more static. Most likely a corrupt packet - it will be dropped instead of the NAK for -EPROTO.

-12 (-ENOMEM)
Out of memory - memory couldn't be allocated for some function; also indicates a GART registration failure (for now)

-14 (-EFAULT)
Bad address - failed RDMA send due to fatal network error

-19 (-ENODEV)
No such device - connection request to invalid device

-53 (-EBADR)
Invalid request descriptor - couldn't post datagram for outgoing connection request

-54 (-EXFULL)
Exchange Full - too many SMSG retransmits

-57 (-EBADSLT)
Invalid slot - datagram match for wrong NID.

-70 (-ECOMM)
Communication error on send - we couldn't send an SMSG (FMA) due to a GNI_RC_TRANSACTION_ERROR to peer. This means that there was some HW issue in trying the send. Check for errors like *SMSG send error to 29@gni: rc 11 (SOURCE_SSID_SRSP:REQUEST_TIMEOUT)* to find the type and cause of the error.

-71 (-EPROTO)
Protocol error - invalid bits in messages, bad magic, wire version, NID wrong for mailbox, bad timeout. Remote peer will receive NAK.

-100 (-ENETDOWN)
Network is down - could not create EP or post datagram for new connection setup

-102 (-ENETRESET)
Network dropped connection because of reset - admin ran *lctl --net gni disconnect*

-103 (-ECONNABORTED)
Software caused connection abort - could not configure EP for new connection with the parameters provided from remote peer

-104 (-ECONNRESET)
Connection reset by peer - remote peer sent CLOSE to us

-108 (-ESHUTDOWN)
Cannot send after transport endpoint shutdown - we are tearing down the LND.

-110 (-ETIMEDOUT)
Connection timed out - connection did not receive SMSG from peer within timeout

-111 (-ECONNREFUSED)
Connection refused - hardware datagram timeout trying to connect to peer.

-113 (-EHOSTUNREACH)

No route to host - error when connection attempt to peer fails

-116 (-ESTALE)

Stale NFS file handle - older connection closed due to new connection request

-117 (-EUCLEAN)

Structure needs cleaning - admin called *lctl --net gni del_peer*

-125 (-ECANCELED)

Operation Canceled - operation terminated due to error injection (fail_loc) - not all injected errors will do this.

-126 (-ENOKEY)

Required key not available - bad checksum

-131 (-ENOTRECOVERABLE)

State not recoverable - stack reset induced TX or connection termination