Stabilizing Lustre at Scale on the Cray XT

Nicholas Henke, *Cray Inc.*

ABSTRACT: *The Cray XT uses Lustre to provide a critical system resource and has provided a unique set of scaling challenges for Lustre. In highlighting the unique attributes of the Cray XT for Lustre, this paper discusses the problems encountered when Lustre is pushed into new frontiers. Using illustrative failure scenarios, I present the mechanisms in Lustre that fail and the root cause of the issues. Solving these problems provides information and insight that will benefit future Cray XT system deployments, especially as the size and complexity of Cray XT systems increases.*

KEYWORDS: XT, Lustre

1. Introduction

With the deployment of large Cray XT systems, much is demanded of Lustre. Lustre provides the parallel file system and I/O capabilities for applications and users. The stability of Lustre is a key system component that has a direct effect on overall system stability and productivity. While the Lustre file system is designed to provide unprecedented metadata and I/O performance rates at the scale of the most demanding high performance computing platforms, [1] pushing it to the very large scale of the Cray XT can result in a myriad of problems.

The Cray XT has pushed the boundaries of system scale and will continue to do so. Increased compute node counts, memory speed and high speed network bandwidth require that the storage subsystem keep pace while maintaining a balanced system configuration. These features of the XT require a unique view on stabilizing system services, demanding that we understand how Lustre operates at scale and the nature of its failures. Only armed with such knowledge can the system as a whole be engineered with Lustre success in mind.

The difficulty in providing a stable Lustre offering on the Cray XT starts with making an order of magnitude jump in scale from Lustre's traditional Linux cluster roots. Increases of this size require that fundamental assumptions are broken and fixing the architecture often requires complex changes. Also, differing customer requirements result in system configuration changes that impact Lustre in a sometimes subtle manner. Lustre itself is a difficult product to administer and triage. Detailed knowledge of the internal workings coupled with the ability to sift through mountains of error messages are needed find the root cause of issues.

It is the unique characteristics of the Cray XT that provide the platform for discovering the weaknesses in Lustre related to scale. The Cray XT is also the only system of its size where a Lustre client is used from every compute node. The scale and density of storage arrays needed to balance processing power are also growing, resulting in Lustre server configurations that also push through previous boundaries. This combination of system attributes delivers a truly impressive load on Lustre. This pressure exposes Lustre code paths that do not scale well and illuminates the vulnerable points in the Lustre implementation.

This paper discusses the framework by which Lustre manages scale related data, the manner in which its reliability mechanisms are exceeded, and how the breakdown of either results in an unstable system. Using real world issues seen while bringing up a Cray XT with over 19,000 compute cores running Catamount and 320 TB of storage, we will take a look at the critical Lustre subsystems involved and how the failures affected the system as a whole. We then discuss how the gained knowledge and understanding of Lustre provides for better system configuration and allows us to look for and find areas of improvement as new hardware, scale and system features progress.

2. Overview of Lustre

Fundamentally, Lustre is internally organized as a distributed client-server network application. Requests are generated on clients, sent to servers and replies are returned. The Lustre servers are responsible for maintaining resource usage and connection data for each of these clients. Each client is represented to the servers at two different levels, Lustre Networking (LNET) at the lower level and Lustre at the top. Each of these layers maintains a request timeout that provides health and peer status.

2.1. Connection representation

At the LNET layer, each network interface is assigned a network ID (NID). On the Cray XT, this translates directly to the NID from Portals, with a process ID being used to differentiate multiple applications on the same processer in Catamount Virtual Node (CVN) and LibLustre. For each of the compute clients, there is a peer-to-peer LNET connection maintained with each of the Lustre server nodes.

At the Lustre layer, the clients represent themselves to the Lustre services as a Universally Unique Identifier (UUID). Each client maintains a connection to each individual Lustre service, namely those of an Object Storage Target (OST) and Metadata Target (MDT). This results in the client keeping multiple service connections on top of each LNET connection.

2.2. Timeouts

To allow for peer and service health detection and other resiliency features, Lustre employs a number of timeouts. These timeouts bound request service times, and exceeding them results in peer or service status changes.

LNET timeouts are used to determine if the underlying network driver and hardware is functioning properly and if the remote peer is responsive at the LNET layer. This means that message traffic is proceeding, and the remote peer is handling that traffic. Exceeding the timeout results in an error message being logged and a return up the stack to Lustre indicating the request could not be serviced.

Lustre timeouts provide service responsiveness information and indicate when a service is down and resiliency mechanisms need to be used to recover that service. There is a core Lustre timeout related to servicing normal I/O requests called *obd_timeout*. On large Cray XT systems, this timeout is set to 300 seconds. The large majority of the timeouts in Lustre use this value directly or reduce it by some factor for quality of service reasons. An example of one such reduced timeout is the timeout that covers initial client to service connection requests with a value of *obd_timeout*/20, or 15 seconds. When Lustre request timeouts are exceeded, Lustre also logs an error and starts retrying the request to the original service and any failover partners configured. The default configuration of Lustre provides those requests will be retried indefinitely until some outside force intervenes.

There are also timeouts covering the server to client communications. Excluding direct responses to client requests, the asynchronous messages sent from a server to client allow resource information gathering and potential revocation. The timeouts covering these asynchronous requests are often quite short as well, allowing a more responsive quality of service and ensuring forward progress when resources are under contention.

3. Scaling challenges

The system configuration is a key factor in determining how I/O load will be distributed across the Lustre servers. With the storage array density increasing, the layout of the storage devices becomes critical. Typically, a storage array is split into several logical devices of which one or more is exported to each Lustre server. Each one of these devices becomes a Lustre OST.

At the time, Lustre required the maximum size of an OST to be 2 TB. With 320 TB of storage, this resulted in 160 OSTs. The system configuration called for 20 Object Storage Server (OSS) nodes, resulting in 8 OSTs per OSS. The machine also has about 9,500 dual core processors, providing a total of 19,000 compute cores and Lustre clients.

This system configuration of 8 OSTs per OSS coupled with the large Lustre client count was the catalyst in very interesting scaling issues with Lustre. The first problem to be discussed is that of RPC load and how this ratio increased that load over the threshold of the Lustre timeouts. The second is the effect on the Lustre servers of trying to process this load and what steps were necessary to ensure that Lustre continued to process requests under extreme resource utilization.

3.1. Lustre RPC load and processing

As discussed in the Lustre overview section, each client maintains a connection for each Lustre service in the configuration. In the case we are using as a running example, this resulted in each client having 8 connections to the Lustre OSTs on each of the 20 OSS nodes, for a total of 160 connections per client. With 19,000 clients, this is 3,040,000 connections in total. Each OSS node saw 152,000 connections.

The initial test application used, simpleio [2], was simple indeed but performed basic operations that simulate a large class of HPC applications. It opened up a unique file per process, wrote and read a small piece of data to verify I/O functionality and then closed the file.

With an application running in the Catamount operating system and the LibLustre client, one of the first actions performed at application launch is the Lustre file system mount. Each client sends a single connect remote procedure call (RPC) to the Lustre services to set up the connections that will be used for file I/O. In the configuration at hand, this is a nearly instantaneous generation of 3,040,000 connect RPCs. The Lustre timeout for each of these requests is 15 seconds, requiring that each OSS process the messages at a rate of 10,133 requests per second to keep from exceeding the timeout.

There were two factors which prevented the Lustre servers from achieving these rates and preventing this simple 19,000 node job from completing. The timeout used for messages was far too short, and the data structures used to store per Lustre service connection information were not scaling well.

3.2. Lustre timeouts

The result of the 15 second Lustre connect timeout being exceeded was the job going into a cycle of RPC retries. These subsequent retries would also timeout, preventing any real forward progress and creating a further backlog on the Lustre servers. The servers did not know the RPCs sitting in its incoming queue had been abandoned by their respective clients and would attempt to process them. This storm of message retries also prevented accurate data from being collected on the server side to help in problem diagnosis. The net effect of this was a loss of the Lustre subsystem as a whole; the servers were saturated to the point where virtually no I/O was progressing.

As is often done when timeouts are exceeded, the action taken was to increase the Lustre *obd_timeout* to 600 seconds and to increase the ratio of Lustre connect RPC timeouts to *obd_timeout*/2 or 300 seconds. This was a time period sufficient to provide data on the server side processing as well as preventing the timeout and retries. The simpleio application was now able to pass through this Lustre service connect phase. Data collected with this new timeout showed that it took 214 seconds to process these incoming connect messages, requiring a further analysis of the issue.

3.3. UUID Data structures

Examination of the profiling data for Lustre server code processing indicated a significant amount of time in routines that verified a client was not already connected. This code was scanning a linked list of client UUIDs and comparing each against the UUID in the connect request. This algorithm results in M/2 comparisons for each connect request, where M is the number of clients already connected. For the 19,000 node system and an application run spanning the entire machine, it generated 180,500,000 total comparisons for each Lustre service or 1,444,000,000 comparisons for each 8 OSTs Lustre server node. Given such a large percentage of time was spent doing these comparisons, it was clear a fix was needed to help reduce the overall connection processing.

The solution for this issue was to use a hash table keyed on the UUID. The hash table is configured with 128 allowable keys, reducing the number of comparisons to M/128. For our example, this now generates 11,017 comparisons per service or 88,134 comparisons per node. Additional improvements were made that allowed simpleio to run on 19,000 cores in under 60 seconds from an initial successful run of 11 minutes.

3.4. Configuration impact

To illustrate the liability of the system configuration in these scaling issues, we compare the connection counts and RPC processing rates for a Cray XT of a similar size. Using our running example configuration as system A, we compare it to system B with 288 TB of storage space split into 144 OSTs on 72 OSS nodes with 23,000 Lustre clients. This system B only required *obd_timeout* tuning to 600s to get simpleio to run.

The configuration of system B generated a total of 3,312,000 connections and 46,000 connections per OST compared to the 3,040,000 total connections and 152,000 connections per OSS for system A. The total number of connections is higher, but the lower OST per OSS ratio spreads them out over more OSS nodes, reducing the individual server load greatly. The RPC processing rate for a Lustre mount also illustrates the load differences. System B needs to be 3,067 requests per second compared to the 10,133 requests per second in system A, over a 3 fold reduction compared to the system in our running example.

In order to provide additional load reduction beyond the Lustre code changes mentioned above, System A was eventually reconfigured with 4 OSTs per OSS. To facilitate this change, Lustre was altered to allow a maximum OST size of 8 TB. This reduced the number of connections per OSS to 76,000 and the RPC processing rate to 5,066 requests per second.

3.5. Second order effects of RPC processing

While the system was able to handle the load from simpleio, running applications that performed large amounts of I/O uncovered other issues. Applications were able to trigger errors in the Portals Lustre Network Driver (LND) indicating that Portals was not servicing requests in the 50 second timeout set by the LND. The investigation into this uncovered a surprising effect of the CPU usage associated with the UUID list traversal and comparisons.

The mechanics of incoming message processing in Lustre largely resembles a pipeline structure. There are three distinct software layers that each take incoming requests interrupts and add them to a queue for

processing. Each layer then uses separate processes to pull a message off the queue and run it through request handling, often resulting in a incoming request interrupt to the layer above it. These three layers are Portals, the Portals LND and LNET.

Timeouts at the Portals LND layer typically mean the remote host has gone away, or there is some problem in the underlying network that is preventing message traffic from flowing. Using the low-level Portals trace information, it was discovered that messages were moving fine and there was no appreciable delay in the network. Further investigation showed the requests were being passed up to and accumulating in the Portals LND queue on the order of tens of thousands of messages. The effect of this backlog of messages was to starve the remote peer of credits by which it could send messages. The aforementioned CPU usage was all happening in the Lustre layer, starving the Portals LND and LNET kernel threads from running and preventing message processing that would have returned credits.

To compound the problem, the Portals LND timeout was actually covering the time it took to send a message, including receiving enough credits from the remote peer and not just the time on the wire. This generated a false indication of the problem, requiring significant data gathering and analysis to realize the misdirection. Addition of a timestamp into the LNET messages when the request is put into and taken out of the Portals layer allows the problem to be seen in much clearer light and prevents underlying layers from being falsely accused.

The fix to this issue required Lustre and not the Portals LND to be altered. Lustre was changed so that it would yield to the process scheduler periodically to ensure the lower layers could process their incoming messages. The Lustre threads, LNET threads and Portals LND threads now share the CPU fairly enough to prevent message backlogs from occurring at any of the layers.

4. Interactions with XT subsystems

While Lustre certainly had internal issues related to scale, it can also cause problems with other system components. This is extremely evident when Lustre is under duress and operating outside normal parameters. The CPU utilization of the Lustre server kernel threads can affect other services running on those nodes. Also, Lustre is widely known to be verbose in its error messages, and the Cray RAS and Management System (CRMS) console network must bear the brunt of these messages.

4.1. Service node heartbeat

The most basic method by which Cray XT systems detect the health of Linux service nodes is through the Resiliency Communication Agent (RCA) heartbeat. The heartbeat service consists of multiple parts, most operating in the CRMS network with a host resident application that provides data to the control network processes. The host resident application is a Linux kernel thread that updates a known location in memory with an increasing value. This value indicates that the node is functioning well enough to schedule processes. The control network processes monitor this location and ensure the value continues to increase. If the value stops incrementing for long enough, a determination is made that the node has collapsed and node death messages are sent. This death is then reflected in the system status commands.

Lustre servers also operate largely through Linux kernel threads. In the particular instance where the CPU was being consumed with UUID list processing, the RCA kernel threads were not scheduled with enough frequency to prevent the node from being marked down. Several workarounds have been used to monitor this false status change and to mark the node back to an operational state by using alternate means of detecting node health. Several Lustre and kernel paths have also been augmented to trigger the heartbeat value increase, helping to keep the node alive even under heavy computation. This remains a tricky issue as several instances have been recorded where the Lustre server is in such a CPU bound state that nothing can prove the node is up and operational and it is declared truly dead.

4.2. Console error messages

The path over which Lustre error messages are passed from both compute and service nodes is via the L0 console and the CRMS network. Lustre is notorious for generating at least one error message when any request exceeds its timeout. Often these timeout messages are accompanied by Lustre service status change messages or other debugging data. Considering the case where 19,000 clients each send 160 connect messages and 75% of them timeout, there are at least 240 messages output per client for a total message count of 4,650,000. These messages are all generated at nearly the same time and the CRMS network as a whole was unable to cope. The resulting loss of the CRMS network disabled the same mechanisms by which the machine can be controlled. The effect on the Cray XT was unique due to a combined console and command-control network.

Certainly improving the CRMS network to cope with misbehaving subsystems is paramount to stability, but Lustre is not free of responsibility. Lustre needs to respect the boundaries of the system and to contain itself during the events that are likely to generate huge volumes of console traffic. An examination of customer console logs was performed to get a list of the worst offending error messages. These messages were reworked or removed entirely. Lustre was also modified to impose an overall rate based limit on the number of error messages to provide a final layer of protection against future storms.

5. Looking forward

Certainly these scaling problems have been challenging, but the outcome has proven satisfactory. Lustre has been stretched to accommodate the large Cray XT configurations and is able to provide a relatively stable system service. Scaling the Cray XT has provided a wealth of information on how Lustre handles load and where it is likely to encounter problems. Lustre timeouts and the operations they cover have been and will continue to be the culprits affecting system stability at scale. Certainly the system configuration can exacerbate the issues, but uncovering the problems is an important step in continuing to improve the Lustre file system. Appreciating the structure of Lustre provides a system level perspective that allows for better understanding of potential issues at scale, and will supply the context through which future systems can be engineered with Lustre in mind.

As the Cray XT continues to evolve, it is important that we continue to look at ways in which Lustre will grow and how we can overcome new challenges. In particular, Compute Node Linux (CNL) changes nearly all of the Lustre client software stack and operating system. CNL brings the Lustre client being used on the Cray XT to that of a Linux based operating system. There are several tradeoffs between Catamount with LibLustre and Linux with the Lustre client that are directly related to scale. By understanding the differences in the Lustre client semantics, we can identify areas of concern with relation to scale.

5.1. Compute Node Linux

In Catamount and LibLustre, the life of the Lustre client is tied directly to the life of the application. Lustre is mounted at application start up and is unmounted at application death. Catamount is unable to take exclusive Lustre locks that live across system call boundaries as the operating system does not provide for interruption and is unable to processes asynchronous events from the Lustre servers in finite time. CNL provides the Lustre client as a system service; the client lives from node boot until the node is shut down. CNL also does not suffer the limitations with respect to asynchronous Lustre requests, allowing locks to be taken on various resources. These locks are a new resource that is distributed across the system, and as such the assumptions around lock behavior and revocation are subject to breaking under the extremes of scale [3].

Lock related RPCs are also covered by timeouts, typically *obd_timeout*/15 or 20 seconds for typically configured Cray XT systems. Given our experience with a similar timeout for Lustre service connect messages, encountering issues in this area is entirely foreseeable. While we can adjust these timeouts to allow for longer request servicing times, we need to ensure that we are not adversely affecting important quality of service guarantees. This becomes even more important as system scale is increased to where component failures become more typical than exception.

All of the Lustre timeouts need to balance between handling load and reasonable detection of node status. A new feature, Adaptive Timeouts [4], is slated to help deal with this delicate balancing act. It should allow base timeouts to be set at a small value to allow rapid health detection but provide a mechanism by which these timeouts can be expanded to cover the actual system needs based on load and observed request response times. It is also with Adaptive Timeouts that Cray might be able to inject external system status into the timeout value providing a path by which we can use RCA heartbeat and other subsystems to enhance Lustre's knowledge of system state.

Although CNL does increase the number of distributed Lustre resources in use, the move to Linux running on the entire multi-core processor provides for a significant reduction in the number of Lustre clients. This reduces the largest scaling component in the client to service connection and server RPC processing rate calculations. For systems with dual core processors, this effectively halves the RPC load on the Lustre servers. Initial system configurations for quad core processors do not show the socket count growing significantly, allowing the Lustre client count to stabilize while still increasing the computing power of the machine.

Acknowledgments

The author would like to highlight the immense amount of help that has enabled this paper. In particular, the wonderful folks in Cray Software Product Support (SPS) and the site analysts have provided immeasurable help with problem diagnosis and data collection. Many folks in Cray Software Development have been instrumental in discussing whole system impact, expected behaviour and potential solutions. The engineers in the Lustre Group at Sun provided in-depth help with data analysis and problem resolution. The author would also like to thank his lovely wife for tolerating him.

References

1. Lustre File system. http://www.sun.com/software/products/lustre

2. Shane Canon, Don Maxwell, Josh Lothian, Kenneth Matney, Makia Minich, H. Sarp Oral, Jeffrey Becklehimer, Cathy Willis "XT7? Integrating and Operating a Conjoined XT3+XT4 System", *CUG 2007*.

3. Peta-Scale I/O with the Lustre file system. http://www.sun.com/software/products/lustre/docs/Peta-Scale_wp.pdf

4. ibid

About the Author

Nicholas Henke is a Software Engineer with Cray Inc. He can be reached by E-Mail at nic@cray.com.