# Lustre Resiliency: Understanding Lustre Message Loss and Tuning for Resiliency

Chris Horn
*Cray Inc.*
*Saint Paul, MN USA*
*hornc@cray.com*

***Abstract*—Cray systems are engineered to withstand the loss of components, however, Lustre, historically, has not been as resilient in some cases. In this paper we discuss recent enhancements made to Lustre to improve resiliency and best practices for realizing Lustre RAS on Cray systems including how to tune timeouts and configure certain Lustre features for resiliency.**

***Keywords*-Lustre; LNet; Resiliency; Availability; Serviceability;**

## I. Introduction

The exchange of requests and replies between hosts forms the basis of the Lustre protocol. One host will send a message containing a request to another and await a reply from that other host. The underlying network protocols used to exchange messages are abstracted away via the Lustre Network (LNet) software layer. The LNet layer is network-type agnostic. It utilizes a Lustre Network Driver (LND) layer to interface with the driver for a specific network type.[1]

Historically, Lustre and LNet have had a poor track record dealing gracefully with the loss of a request message or the associated reply. This difficulty is largely due to their reliance on internal timers and per-message timeouts to infer message loss and the health of participating hosts. In addition, there has long been a fundamental single point of failure in the Lustre protocol whereby Lustre servers would not re-send certain requests to clients which could result in application failure.

Cray has worked with Seagate and the Lustre open source community to address the flaw in the Lustre protocol, fix new issues that were discovered as part of that effort, and to tune Lustre, LNet, and LND timers and timeouts to maximize the resiliency of Lustre in the face of message loss. Changes are still landing to the canonical tree, but we expect Lustre 2.8.0 to be fully resilient to lost traffic such that clients can survive finite network disruptions without application failure and message loss has minimal performance impact.

This paper covers a number of topics as they relate to Lustre resiliency. To provide some background we first discuss the role of Lustre evictions and locking in a Lustre file system, the effects of message loss, and Lustre's response to some common component failures. Finally, we discuss our recommendations for tuning Lustre to realize improved resiliency on Cray systems.

## II. Understanding Lustre Client Evictions

An eviction in Lustre is an action taken by a server when the server determines that the client can no longer participate in file system operations. Servers take this action to ensure the file system remains usable when a client fails or otherwise misbehaves. When a client is evicted all of its locks are invalidated. As a result, any of the client's cached inodes are invalidated and any dirty pages must be dropped.

Servers can evict clients if clients do not respond to certain server requests in a timely manner, or if clients do not communicate with the server at regular intervals. Evictions of the latter sort are carried out by a server feature called the ping evictor.

The ping evictor exists to prevent the problem of cascading timeouts [2]. Since message timeouts are used to determine a connection's state a problem occurs where dependencies between requests result in implicit dependencies between connections. If a client holds a resource and dies the server would not realize this until a conflicting request for the resource is made. The server would then need to wait for a message timeout to detect the failed client. This can cascade as the number of resources held by a client increases or the number of dead clients increases. The ping evictor helps with this problem by proactively detecting failed clients and reclaiming their resources via eviction.

Idle clients are expected to communicate with Lustre servers at regular intervals by sending servers a Portal RPC (PtlRPC) ping. The interval is equal to `obd_timeout / 4` where `obd_timeout` is a configurable Lustre parameter and defaults to 100 seconds for a 25 second ping interval (Note, Cray has historically used a 300 second `obd_timeout`). The ping evictor on a server keeps track of these client pings. If a particular client has failed to deliver a ping within `1.5*obd_timeout` seconds[1], and the server has not seen any other RPC traffic from that client, then the ping evictor will evict the client.

Clients can exhibit misbehavior for a number of reasons including: client side bugs, a kernel panic or oops, heavy

[1]Servers are very conservative about evicting clients. Up to two pings may be missed or lost.

load, and serious Lustre errors known as LBUGs. In addition, component failures external to the client may result in client evictions. We'll discuss this topic in more depth in section V.

A client will learn that it has been evicted the next time it connects to the server. When the client learns of its eviction it must drop all locks since they've been invalidated by the server, and it must drop all dirty pages since it no longer has the requisite locks.

Clients may be unaware of their eviction if they do not have any outstanding user request and previous requests were buffered. This behavior is POSIX semantics, so applications need to check return codes or use fsync(2). Unaware users may call this silent data corruption.

## III. Understanding Lustre Locking

In Lustre, shared resources are protected by locks, and the most common resource is a file. When an application wishes to read or modify a file in some way it contacts the Metadata Server (MDS) for the `open()` system call. The MDS provides striping information for the file. The client will then enqueue a lock for each stripe of the file to the respective Object Storage Target (OST) which are hosted by Object Storage Servers (OSS). The OSS checks whether the lock request conflicts with any other lock that has already been granted, and sends a blocking callback request to any client holding a conflicting lock. The response to this blocking callback must, eventually, be a lock cancel.

Once all conflicting locks have been cancelled a completion callback request is sent to the enqueuing client which grants the lock. The client must acknowledge receipt of the completion callback before it may start using the lock.

There are three lock related requests that are only ever sent from servers to clients. Two are the blocking and completion callback requests mentioned earlier, and the other is the glimpse callback request. These RPCs are sometimes referred to as Asynchronous System Trap (AST) RPCs or, more commonly, as just ASTs.

When a server sends a blocking AST to a client, either as a standalone message or embedded within a completion AST, the server sets a timer after which the client shall be evicted if it has not fulfilled the server's request. This is referred to as the lock callback timer.

The blocking AST informs the client that a previously granted lock must be cancelled. The client must acknowledge receipt of the blocking AST, finish any I/O occurring under the lock, and then send a lock cancel request to the server. If the lock has not been cancelled by the time the lock callback timer has expired then the client shall be evicted. The server will refresh the lock callback timer as the client performs bulk reads and writes under the lock, so the client is afforded sufficient time to complete its I/O.

To illustrate why evictions are necessary consider two clients that want access to the same file. The first client, the writer, creates a file, writes some data to the file, and then promptly suffers a kernel panic. The second client, the reader, wants to read the data written by the first client. The reader requests a protected read lock for each stripe of the file from the corresponding OST. Upon receipt of the lock enqueue request, the OSS notes that it has already granted a conflicting lock to the writer. As a result the server sends a blocking AST to the writer, and it arms a lock callback timer for the lock. Since the writer has crashed it will not be able to fulfill the server's request. At this point, the server cannot grant the lock to the reader, and so the reader is unable to complete its `read()`. When the lock callback timer expires the server will evict the writer which will invalidate all of the writer's locks, and, since the writer no longer holds a valid lock on the file, the server can grant the lock to the reader.

## IV. Dropped RPCs and Lustre

When a Lustre RPC is lost users may observe performance degradation or, in the worst case, client eviction and application failure. In this section we describe some of the different issues that Lustre must deal with in order to recover successfully from an RPC loss.

### A. Detecting Message Loss With RPC Timeouts

When a client sends an RPC to a server, or a server sends an RPC to a client, a timeout value is assigned to the RPC. This timeout value is derived from Lustre's Adaptive Timeouts feature, and it accounts for the time it takes to deliver the RPC (network latency) and the amount of time the message recipient needs to process the request and send a reply (service time). A host infers that a message has been lost when this timeout expires and the host has not received a response from the message recipient. This is in contrast to a failure to send an RPC in the first place which can typically be detected much sooner because LNDs will typically notify upper layers when they are unable to send an RPC.

When an RPC timeout occurs the connection between the sender and recipient is severed and the client must reconnect to the server by sending a connect RPC. When a connection is reestablished with the server the client must resend the lost RPC. The client must do all of this without repeating the cycle of lost RPCs, disconnections, reconnections, and resends. In a routed environment this means we need to avoid using any bad routes which may have been the culprit in the initial lost RPC, or in the event of an HSN quiesce we would ideally only try to send messages once the quiesce was over. In the case of a lost lock cancel, or a reply to blocking AST, the client must do all of this as quickly as possible so as to avoid the lock callback timer expiration on the server and subsequent eviction.

More information on Adaptive Timeouts is available in VII-A.

### B. Avoiding Bad Routes

When messages are dropped due to route or router failure any LNet peer using that route or router, i.e. a Lustre client or server, may be impacted, and since there are typically many more clients than routers we can expect a relatively large disruption from a single router failure. Using bad routes wastes time and resources because there are a limited number of messages that a peer can have in flight and any message sent over a bad route will need to be resent. Thus, it is very desirable to detect bad routes as quickly as possible and remove them from an LNet peer's routing table. Traditionally route and router health is determined by the LNet Router Pinger and Asymmetric Route Failure Detection features. These features work in conjunction to determine the health of both routers themselves and the routes hosted by routers.

The router pinger on an LNet peer works by periodically sending an LNet ping to each known router. If a peer receives a response from the router within a timeout period then the router is considered alive.

The Asymmetric Route Failure Detection feature works by packing the router's ping reply with additional information about the status, up or down, of the router's network interfaces. When an LNet peer receives a ping reply it inspects the network interface status information to determine for which remote networks the router should be used. If a remote interface on the router is not healthy then that router will not be used when sending messages to peers on the associated remote network.

See section V for a description of how these features are used to respond to LNet router and route failure. For more information on tuning the LNet router pinger and asymmetric route failure detection features see VII.

### C. Reconnecting with the Connect RPC

The first thing a client must do after experiencing an RPC timeout is reestablish a connection with the target of the lost RPC. This is accomplished via the connect RPC. Lustre targets can be reached via multiple LNet nids for the purposes of failover or a multi-homed server. Connect RPCs are first sent using the LNet nid of the last known good connection before trying any alternative nids in a round-robin manner. Connect RPCs are sent on an interval, so if one connection attempt is unsuccessful there may be a delay until a client attempts the next one.

This is a significant complicating factor in a client's ability to reestablish connection with the server in a timely manner. Consider the case of message loss due to a failed route. During the time it takes to detect the failed remote interface the client considers the bad route to be valid and will continue to use it for sending RPCs to the servers. If it happens to send the connect RPC using the bad route then this RPC will eventually time out. As a result the client will send subsequent connect RPCs to alternate nids where the Lustre targets may not be available. These connection attempts will be rejected, typically with `-ENODEV`, as those resources are not available at that host. This delays the process of reestablishing the connection even further.

We can see that the timing of our reconnect attempts is important. The faster we reconnect the more time we'll have to resend important RPCs to mitigate performance impact or avoid eviction, however, the faster we reconnect the more likely we are to hit a bad route or attempt to send while the network is otherwise unable to reliably deliver messages.

### D. Resending Lost RPCs

The final action a client must take is to resend any RPCs it thinks were lost. As with the connect RPC, the client must take care to avoid bad routes when resending lost RPCs to avoid repeating the lost RPC cycle. Lustre clients have long had the ability to resend bulk RPCs, but not the ASTs described in section III.

Lustre's inability to resend ASTs was a single point of failure in the Lustre protocol that could result in client eviction whenever an AST was not delivered, or the reply to an AST was lost. We'll discuss the solution to this problem in VI.

## V. Failure Scenarios

There are a number of scenarios under which Lustre messages may be lost. In this section we discuss some of the more common scenarios and Lustre's response.

### A. LNet Routing

In this section we consider two resiliency issues specific to LNet routing: LNet router node death, and the death of a remote interface on an LNet router.

*1) Router death:* If a router node dies, e.g. because it suffered a kernel panic, then any messages in transit to that router and any messages buffered on that router at the time it died will need to be resent. The LNet layer does not track which routers are used to send particular messages, and the Portal RPC (PtlRPC) layer does not have access to this information either. As a result, PtlRPC and LNet do not know to try a different router when a previous send failed.[2]. We thus rely on the router pinger to determine router aliveness. If a router ping was in flight at the time of the panic then we should be able to mark the router down after the ping timeout has expired. Otherwise we may need to wait for the next ping interval in addition to the time needed for the ping to timeout in order to mark the router as down. During the time between the kernel panic and marking of the router as down the router can still be used as a next-hop.

[2]All routes to a remote network are used in a round-robin manner

*2) Remote Interface Death:* As discussed in IV-B, LNet peers rely on the Asymmetric Route Failure Detection feature to determine the health of remote interfaces on an LNet router node. For example, say a router has one Infiniband (IB) interface configured on LNet o2ib0, another infiniband interface configured on LNet o2ib1, and one Aries interface on LNet gni0. The LNet nids for each of these interfaces are 10.100.0.0@o2ib0, 10.100.1.1@o2ib1, and 27@gni respectively.

A client on the gni LNet has two routes that utilize 27@gni for the remote networks o2ib0 and o2ib1. i.e. if the client wants to send an RPC to a server on either o2ib0 or o2ib1 it can use 27@gni as a next-hop for those messages.

Now suppose the router's IB interface at 10.100.1.1@o2ib1 fails. After some period of time, detailed below, the router marks that interface as down. The client's router pinger sends a ping to the router at the next ping interval. Since the router node is up, and the gni interface is healthy, the router will respond to the client with the information that 10.100.0.0@o2ib0 is up and 10.100.1.1@o2ib1 is down. The client sees that this router's interface for the o2ib1 network is down, so it will no longer use 27@gni as a next-hop when it needs to send messages to servers on the o2ib1 LNet. However, it will continue to use 27@gni as a next-hop when it needs to send messages to servers on the o2ib0 LNet.

When the router's interface on o2ib1 eventually recovers the router will mark that interface as alive as soon as it sees traffic come over that interface.[3] The next router ping from the client will retrieve the updated information, and the client will then begin using 27@gni as a next-hop for sending messages to peers on the o2ib1 LNet.

It should be noted that it takes a significant amount of time to propagate router health information to peers, and it takes additional time to propagate a change in remote interface health to peers. Routers infer local interface health by monitoring traffic over the interfaces. The router is aware that peers will be sending router pings, and the longest interval at which these pings will occur. Thus, if an interface does not receive any traffic in this interval, plus the timeout value for router pings, then the router will assume the underlying interface is not healthy and will change its status to down.

Using Lustre's default values, it takes a router 110 seconds, based on a 60 second ping interval plus a 50 second ping timeout, to mark an interface down. As mentioned previously, peers only learn about the interface status change after pinging the router. With a 60 second ping interval and 50 second ping timeout the worst case to detect a failed remote interface is on the order of 220 seconds. In practice the worst case isn't quite as bad if the local network is healthy since the peer should get a ping reply quickly. Thus, it is generally closer to the time for the router to detect the failed interface plus the ping interval or approximately 170 seconds.

[3]Peers on the o2ib1 LNet will be sending LNet pings to the "down" interface at an interval specified by the dead_router_check_interval LNet module parameter.

### B. Client Death

When a client dies it will eventually be evicted so that any resources held by that client might be reclaimed. This will be accomplished either via a lock callback timer expiration or via the ping evictor. If the client holds a conflicting lock then it may be evicted by a server issuing a blocking callback for that lock. Otherwise it will eventually be evicted by all Lustre servers when it fails to ping them after one and a half times the `obd_timeout`.

### C. Server Death

When a server dies in a high availability (HA) configuration its resources (Lustre targets) should failover to its HA partner. The Lustre recovery feature should generally ensure the filesystem returns to a usable state. For more information on Lustre recovery see [3].

### D. Link Resiliency

Cray systems are engineered to withstand the loss of certain components without requiring a system reboot. The same technology is used to allow manual removal and replacement of compute blades without a system reboot (warm swap). This technology is collectively known as link resiliency.[4]

Traffic on a Gemini or Aries HSN is routed over what are termed links. A link is a software term for a connection between two Gemini or Aries Link Control Blocks (LCBs). A software daemon runs on each blade controller that is able to detect failed links as well as power loss to a Gemini or Aries Network Card. When a link failure is detected this daemon sends an event which is received by a software daemon, `xtnlrd`, running on the System Management Workstation (SMW). The `xtnlrd` daemon is responsible for coordinating the steps needed to recover from the failure. The `xtnlrd` daemon also coordinates the steps needed to fulfill a warm swap request.

At a high level the steps needed to recover from link failure include computing new routes that do not use the failed links, quiescing the network, installing the new routes, and unquiescing the network. It is necessary to quiesce the HSN in order to avoid inconsistent routing of traffic, dead ends, or cycles when installing the new routes.

Lustre servers have no knowledge of an HSN quiesce. While the HSN is quiesced servers will not see any traffic from clients, and they will not be able to deliver messages to clients. If clients do not resume communication with servers in a timely manner, i.e. before the expiration of any lock callback timers or the ping eviction timer, then they will be evicted. Servers cannot distinguish between a

client caught up in an extended network outage and a client that has crashed. Thus, it is crucial that clients, servers, and routers do whatever they can to restore connectivity and resume communication as quickly as possible. This includes ensuring that the lock callback timer accommodates this time span.

Another side effect of the HSN quiesce is that LNet routers will be unable to respond to pings from a client's router pinger. This can result in routers being marked dead and removed from the routing tables of the clients. If all routers are marked down in this manner then client's will be unable to send messages to servers until the router pinger has been able to mark routers back up.

## VI. Resiliency Enhancements

A number of enhancements have been made to improve Lustre resiliency in the face of message loss. The primary enhancement is the ability for servers to resend ASTs. Additional enhancements were made to improve the success rate of resending ASTs and avoid lock related evictions, as well as improvements to file system performance in the face of message loss. In this section we discuss these new features and improvements. In the following section we discuss how to tune Lustre for resiliency.

### A. LDLM Resend

As discussed in IV-D, Lustre servers historically did not resend ASTs. This meant that if an AST was lost then the target of that AST would almost certainly be evicted. LDLM resend is the primary enhancement made to fill this hole in Lustre's protocol.

Before the LDLM resend enhancement was implemented, if a blocking or completion callback RPC was lost, because it was, say, buffered on a router when the router suffered a kernel panic, then the recipient of the RPC would be evicted once the lock callback timer expired even though it never received the RPC.

The LDLM resend enhancement allows servers to resend these callback RPCs throughout the duration of the lock callback timer. Callback RPCs, like most other RPCs in Lustre, are assigned a timeout based on the adaptive timeouts feature. If that timeout expires, and a reply has not been received, then the server will resend the callback RPC.

See VII-B for information on tuning the lock callback timer to allow the LDLM resend enhancement to function properly.

### B. Router Failure Detection

Cray's LND for its Aries HSN, gnilnd, has previously utilized health information available on our high speed network to inform routers about the aliveness of peers. This is referred to as the peer health feature for routers. We've recently extended our use of this health information to clients. With this enhancement clients now receive an event when a router node fails. Upon receipt of this event gnilnd will notify LNet that the router is no longer alive and thus remove it from the client's routing table. This provides us another agent, in addition to the router pinger, which can remove bad routes from our routing tables and potentially do so faster than relying on the router pinger.

### C. Fast Reconnect

Lustre clients on the aries HSN have some knowledge of an HSN quiesce due to the gnilnd. The gnilnd will participate in quiesce by suspending all transmits until the quiesce is over. While the quiesce is ongoing, connections between gnilnd peers can timeout. Historically, gnilnd on a client would only attempt to reestablish a connection with a peer (router) when an upper layer generated a request. We recently added a `fast_reconnect` feature, which will force gnilnd on a client to quickly reconnect to routers as soon as the quiesce is lifted. When a connection is established, gnilnd will notify LNet, which will ensure the router is considered alive and can be used as a next-hop for future sends.

### D. Minimizing Performance Impact

Cray has worked to minimize the performance impact of message loss and resiliency events by fixing bugs and employing defensive programming.. An example of the former is a bug found in the early reply mechanism which resulted in RPC timeouts being extended much longer than they should be, and an example of the latter is an upstream enhancement we've recently adopted which places a limit on maximum bulk transfer times separate from the hard limit defined for adaptive timeouts.

*1) Early Replies:* Early reply is a feature that allows servers to request that clients extend their deadlines for RPCs. Any RPC queued on a server for processing is eligible for early replies. The early reply mechanism chooses requests that are about to expire, but still queued for service, and sends early replies for those requests to extend the deadlines.

While testing Lustre resiliency in the face of LNet router loss we found a bug in the early reply algorithm for determining the extended deadline. This bug resulted in deadlines set well beyond the maximum allowable service time (see at_max in Section VI. A.). In one instance we noticed timeouts on servers of 1076 seconds, and associated timeouts on clients of over 1300 seconds when the maximum service time was configured to be 600 seconds. With the bug fixed we were able to emplace effective bounds on maximum service times and limit the performance impact of RPC loss due to router failure.

*2) Capping Bulk Transfer Times:* A bulk transfer in Lustre is performed for reads and writes. In either case a client sends a bulk RPC request to the server describing the transfer. The server then processes the request, performs any

necessary work to prepare for the transfer, and then initiates the data transfer either from the client for writes or to the client for reads.

Historically, bulk transfer times, i.e. the time to perform the actual data transfer once all prerequisite work had finished, were bounded by the maximum adaptive timeout. Bulk data transfer messages are not resent, so when one is lost there is no point in waiting such a long time. A change was made to configure a static timeout for bulk transfer separate from adaptive timeouts.

### E. Peer Health

The LNet peer health feature is not a recent enhancement, but it can play an important role in Lustre resiliency for routed configurations. This feature can assist in efficiently failing messages that are sent to dead peers. When this feature is enabled, prior to sending traffic to a particular peer, LNet will query the interface the peer is on to determine whether the peer is alive or dead. If the peer is dead then LNET will abort the send. This helps us avoid attempts to communicate with known dead peers. Communicating with dead peers wastes resources including network interface credits, router buffer credits, etc., that could otherwise be used to communicate with alive peers. This feature can be used for messages sent to both Lustre clients and servers. This feature should only be enabled on LNet routers otherwise it can interfere with the LNet router pinger feature by dropping the router pings being sent from clients and servers to LNet routers.

## VII. Tuning Lustre for Resiliency

There are a number of tunable parameters that can affect the performance of Lustre on Cray hardware during and after resiliency events. The goal is to survive transient network failure without suffering any client evictions and with minimal impact on application performance. As the discussion in IV outlined, there are a number of areas to consider in the face of RPC loss. Our tuning recommendations strive to strike a balance between the competing priorities of avoiding client evictions where possible while maintaining the ability to detect misbehaving clients in a reasonable time frame.

### A. Adaptive Timeouts

In a Lustre file system servers keep track of the time it takes for RPCs to be completed. This information is reported back to clients who utilize the information to estimate the time needed for future requests and set appropriate RPC timeouts. Minimum and maximum service times can be configured via the `at_min` and `at_max` kernel module parameters, respectively.

*1) at_min:* This is the minimum processing time that a server will report back to a client. Note, it is not actually the minimum amount of time a server will take to process a request. For Lustre clients on an Aries or Gemini HSN Cray's recommendation is to set this to 40 seconds. When an RPC is lost we want it to timeout quickly so that we can resend it and minimize performance impact and avoid client eviction. However, we also want to avoid unnecessary timeouts due to transient network quiesces. The 40 second value factors into our calculation for an appropriate LDLM timeout as discussed in section VII-B

Our recommendation for Lustre servers is also 40 seconds.

*2) at_max:* This is the maximum amount of time that a server can take to process a request. If a server has reached this value then the RPC times out. For Lustre clients on an Aries or Gemini HSN Cray's recommendation is to set this to 400 seconds. Our goal with this value is to provide servers ample time to process requests when they are under heavy load, but also limit the potential worst case I/O delay for requests which will not be processed.

The worst case I/O delay on a client resulting from message loss, assuming the underlying network recovers and is healthy, is equal to the largest potential RPC timeout that a client can set. Since clients must account for network latency to and from a server, in addition to server processing time, the largest potential RPC timeout is larger than `at_max`. In fact, Lustre uses the same adaptive timeout mechanism to track and estimate network latency with the same lower and upper bounds as service time estimates. Thus, the largest potential RPC timeout that a client can set is `2*at_max`. By lowering `at_max` from 600 to 400 seconds we reduce the worst case I/O delay from 1200 seconds, or 20 minutes, to 800 seconds or just over 13 minutes.

Our recommendation for Lustre servers is also 400 seconds.

### B. LDLM Timeouts

The timeouts for LDLM RPCs use the same adaptive timeout mechanism as other RPCs, however the lower bound for the server's lock callback timer can be configured via the `ldlm_enqueue_min` parameter. Per our previous discussion we know that servers must afford the client enough time to timeout a lost RPC, reconnect to the target of the lost RPC, and resend the lost RPC. In addition, since no traffic flows during an HSN quiesce we need to account for the time spent in, and time to recover from, a quiesce as well.

We also know that lock callback timers are used to prevent misbehaving clients from hoarding resources and hindering file system usability, so we need to balance between the competing goals of allowing clients and servers enough time to recover from a network outage (larger `ldlm_enqueue_min`) and quickly detecting misbehaving clients (lower `ldlm_enqueue_min`).

Figure 1 shows the variables which must be considered in setting an appropriate `ldlm_enqueue_min`.

As mentioned in the previous section, both network latency and RPC service times have a lower bound of

$$ldlm\_enqueue\_min = max(2 * net\_latency, net\_latency + quiescent\_time) + 2 * service\_time$$

Figure 1. Equation for ldlm_enqueue_min

at_min. The quiescent_time in this formula is to account for the time it takes all Lustre clients to reestablish connections with all Lustre targets following an HSN quiesce. We've experimentally determined an average time to be approximately 140 seconds, but it is possible that this value may vary based on different factors such as the number of Lustre clients, the number of Lustre targets, the number of Lustre file systems mounted on each client, etc. Thus, given an at_min of 40 seconds, we calculate an appropriate ldlm_enqueue_min as:

$$\begin{aligned} ldlm\_enqueue\_min &= max(2 * 40, 40 + 140) + 2 * 40 \\ &= 180 + 80 = 260 \end{aligned}$$

The value for Lustre servers should match that of clients.

### C. Router Pinger

The goal of tuning the router pinger is to quickly detect bad routes and routers, so that they can be removed from routing tables. The interval at which LNet peers send pings to routers and the timeout value of each ping are configured via kernel module parameters. The ping interval can be configured separately for live and dead routers. The live_router_check_interval specifies the time interval after which the router pinger will ping all live routers, and the dead_router_check_interval specifies the same only for dead routers. The router_ping_timeout parameter specifies how long a peer will wait for a response to a router ping before deciding the router is dead. The router_ping_timeout should generally be no less than the LND timeout.

*1) Servers:* Since there are typically fewer servers than routers we can safely use a more frequent router ping interval and a lower ping timeout on external servers. In addition, since external servers are unable to subscribe to node failure events in the same fashion as Lustre clients on the HSN it is import to lower the ping interval and timeout so that servers can detect failed routers more quickly. Cray recommends setting both the live and dead router check interval to 35 seconds, and the router ping timeout to 10 seconds.

*2) Clients:* Since there are typically many more clients than routers we recommend using the default ping interval and timeout to avoid overwhelming routers with pings. The default values for Cray's Lustre clients are 60 second ping intervals for both live and dead routers, and a 50 second ping timeout. The ping interval can be increased further for very large systems.

*3) Routers:* While routers usually do not ping other routers they do need to be aware of when to expect pings from other peers. Specifically, they need to be aware of the longest ping interval and timeout, so that they can detect when a network interface is malfunctioning as described in V-A2. Thus, the router's live_router_check_interval should be equal to the maximum of the server's live_router_check_interval and the client's live_router_check_interval. The same holds for the router's dead_router_check_interval and router_ping_timeout.

### D. Asymmetric Route Failure Detection

The asymmetric route failure detection feature is enabled by default starting in Lustre 2.4.0. It can be explicitly enabled or disabled via the avoid_asym_router_failure LNet module parameter.

### E. Lustre Network Driver Tuning

*1) Servers and Clients:* Internal testing revealed the default ko2iblnd timeout is unnecessarily high. Cray recommends lowering the ko2iblnd timeout and ko2iblnd keepalive parameters to 10 and 30 seconds respectively, so that the o2iblnd can better detect transmission problems. As discussed in VI-E, the peer health feature should be disabled by setting ko2iblnd peer_timeout=0 and kgnilnd peer_health=0.

*2) Routers:* Internal testing revealed the default ko2iblnd timeout is unnecessarily high. Cray recommends lowering the ko2iblnd timeout to 10 seconds, so that the o2iblnd can better detect transmission problems. The peer health feature should be enabled for the o2iblnd by setting peer_timeout equal to the sum of the server's ko2iblnd timeout and ko2iblnd keepalive, or 40 seconds, and for kgnilnd by setting kgnilnd peer_health=1.

### F. Parameters by Node Type

This section provides an overview of all our recommended parameters by type of Lustre node. This is intended as a reference for the recommendations laid out in this paper, and not a comprehensive guide to all module parameters needed for a functional Lustre filesystem. Please refer to S-0010-5203 in Craydoc for complete documentation.

Figure 2 contains a sample modprobe configuration file for an LNet router node. Figure 3 contains a sample modprobe configuration file for a Sonexion or CLFS Lustre server. Figure 4 contains a sample modprobe configuration

file for a Lustre client on an Aries or Gemini HSN, and figure 5 contains a sample modprobe configuration file for a CDL client.

```
options ko2iblnd timeout=10
# Server's ko2iblnd timeout +
# Server's ko2iblnd keep_alive
options ko2iblnd peer_timeout=40
options kgnilnd peer_health=1

# max(server's router_ping_timeout,
#     client's router_ping_timeout)
options lnet router_ping_timeout=50
# max(server's live_router_check_interval,
#     client's live_router_check_interval)
options lnet live_router_check_interval=60
# max(server's dead_router_check_interval,
#     client's dead_router_check_interval)
options lnet dead_router_check_interval=60
```

Figure 2. Sample modprobe.conf for an LNet router

```
options ko2iblnd timeout=10
options ko2iblnd peer_timeout=0
options ko2iblnd keepalive=30

options lnet router_ping_timeout=10
options lnet live_router_check_interval=35
options lnet dead_router_check_interval=35
options lnet avoid_asym_router_failure=1

options ptlrpc at_max=400
options ptlrpc at_min=40
options ptlrpc ldlm_enqueue_min=260
```

Figure 3. Sample modprobe.conf for a Sonexion or CLFS server

```
options kgnilnd peer_health=0

options lnet router_ping_timeout=50
options lnet live_router_check_interval=60
options lnet dead_router_check_interval=60
options lnet avoid_asym_router_failure=1

options ptlrpc at_max=400
options ptlrpc at_min=40
options ptlrpc ldlm_enqueue_min=260
```

Figure 4. Sample modprobe.conf for an Aries or Gemini Lustre client

## VIII. Site-Specific Tuning

The recommendations laid out in VII were shown to eliminate client evictions from LNet route and router failure,

```
options ko2iblnd timeout=10
options ko2iblnd peer_timeout=0
options ko2iblnd keepalive=30

options ptlrpc at_max=400
options ptlrpc at_min=40
options ptlrpc ldlm_enqueue_min=260
```

Figure 5. Sample modprobe.conf for an CDL client

as well as link resiliency events, on a 19 cabinet XC30, but it is unlikely that we can provide one set of settings for every system configuration. Thus we should consider ways in which these parameters may need to be modified.

Based on our understanding of Lustre's resiliency features, the Lustre protocol, and the effects of the different tunable parameters, we can reason about how the parameter settings might be tweaked to maintain resiliency under different conditions and system configuration. Relevant factors may include routed vs. non-routed filesystems, scale, workload, and network technology.

For example, if server performance profiling indicates that servers are rarely, if ever, under heavy load then `at_max` might be lowered to further reduce worst case client side timeouts. Similarly, if servers are frequently under high load it may be desirable to increase `at_max` to allow servers additional time to process requests and avoid unnecessary RPC timeouts. Figure 6 displays a Lustre error message which may indicate the need to increase `at_max` to allow servers additional time to process requests.

```
Lustre: ost_io: This server is not able to
 keep up with request traffic (cpu-bound).
```

Figure 6. Sample error message indicating slow request processing

It is also likely that the lock callback timer will need to be adjusted to account for system configuration. We can look at certain messages that appear in the console log as a starting point for determining the length of the quiescent time in determining an appropriate `ldlm_enqueue_min`. Figure 7 shows messages printed to the console log as part of a link resiliency event. The message originate from a single client, and they indicate the beginning of the resiliency event as well as when the client was able to reconnect to the Lustre targets. The timestamps indicate that it took this client 125 seconds to recover from the link resiliency event.

## IX. Future Work

Cray is continually working to improve Lustre resiliency. This section provides a brief look at some of our current work and future plans.

In order to enhance our ability to test Lustre resiliency, we've developed a Network Request Scheduler (NRS) policy

```
21:26:51.388273-05:00 c1-0c2s5n0 LNet: Quiesce start: hardware quiesce
21:27:06.393195-05:00 c1-0c2s5n0 LNet: Quiesce complete: hardware quiesce
21:27:13.429388-05:00 c1-0c2s5n0 LNet: Quiesce start: hardware quiesce
21:27:23.435159-05:00 c1-0c2s5n0 LNet: Quiesce complete: hardware quiesce
21:28:24.938501-05:00 c1-0c2s5n0 Lustre: snx11023-OST0009-osc-ffff880833997000: Connection restored to
snx11023-OST0009 (at 10.149.4.7@o2ib)
21:28:49.952123-05:00 c1-0c2s5n0 Lustre: snx11023-OST0002-osc-ffff880833997000: Connection restored to
snx11023-OST0002 (at 10.149.4.5@o2ib)
21:29:05.252357-05:00 c1-0c2s5n0 Lustre: snx11023-OST000c-osc-ffff880833997000: Connection restored to
snx11023-OST000c (at 10.149.4.8@o2ib)
```

Figure 7. These messages indicate 124 seconds for a client to recover from a link resiliency event

for fault injection. The NRS policy, termed NRS Delay, allows us to simulate high server load in resiliency testing by selectively introducing delays into server side request processing. We'll be sharing this work with the community in `https://jira.hpdd.intel.com/browse/LU-6283`

Another area we'd like to improve is in our ability to control RPC timeouts. As discussed in VII-A, the worst case RPC timeouts are `2*at_max`. This is because an RPC timeout is the sum of the estimated network latency and estimated service time. The upper and lower bounds of both of these estimates cannot currently be configured separately. If we were able to configure them separately then we might be able to lower the worst-case timeouts further.

We're also working on infrastructure to remove Lustre's dependence on the ping evictor to maintain client connections. This has two primary benefits. One, we'll be able to eliminate Lustre client pings which can be a source of jitter and I/O disruption [5]. Secondly, we will be able to reclaim resources from failed clients more quickly.

Lastly, work on this paper has revealed that we should be able to create guidelines for site specific tuning. It is unlikely that we can determine a one size fits all solution for configuring Lustre, so providing guidelines for determining appropriate configuration settings is crucial to ensuring every system has optimal resiliency.

## X. Conclusion

Historically, Lustre has done a poor job of handling message loss in a graceful manner. A flaw in the Lustre protocol would often result in client eviction whenever certain RPCs were lost, and lost messages could also cause performance degradation. Cray has worked with our support vendor and the Lustre open source community to address the flaw in the Lustre protocol and minimize the performance impact of message loss.

The LDLM resend enhancement has addressed the flaw in the Lustre protocol, while additional enhancements, such as those made to LNet router failure detection and fast gnilnd reconnects, have helped increase the effectiveness of LDLM resend while also helping to mitigate the performance impact of message loss.

The tuning recommendations and best practices laid out in this paper work in concert with these new capabilities to achieve improved Lustre and LNet resiliency.

## Acknowledgment

Cray would like to thank the Lustre engineers at Seagate who were instrumental in development of the LDLM resend feature. We would also like to thank the Lustre open source community for their contributions and code review, and our customers who push us to improve the state of the art in Lustre.

## References

[1] F. Wang, S. Oral, G. Shipman, O. Drokin, T. Wang, and I. Huang. *Understanding Lustre Filesystem Internals* Oak Ridge National Laboratory, National Center for Computational Sciences, Tech. Rep., 2009, technical Report ORNL/TM-2009/117.

[2] Peter Braam, Alex Tomas. (2008, February 9). *The cascading timeouts problem and the solution* [Online]. Available: http://wiki.lustre.org/images/f/f0/Cascading-timeouts-hld.pdf

[3] Intel. (2015). *Lustre Operations Manual* [Online]. Available: https://wiki.hpdd.intel.com/display/PUB/Documentation

[4] Cray. (2014, October). *Network Resiliency for Cray XC Systems S-0041-C* [Online]. Available: http://docs.cray.com/books/S-0041-C/S-0041-C.pdf

[5] C. Spitz, et. al. (2012). *Minimizing Lustre Ping Effects at Scale on Cray Systems* [Paper presented at the Cray User Group conference, Stuttgart, Germany, April 29 - May 3, 2012].