LNet Health HLD

Target release	Lustre 2.12
Epic	**LU-9120** - Getting issue details... STATUS
Document status	DRAFT
Document owner	Amir Shehata
Designer	Amir Shehata Olaf Weber
Developers	Amir Shehata
QA	Amir Shehata

Adding Resiliency to LNet

Introduction

By design LNet is a lossy connectionless network: there are cases where messages can be dropped without the sender being notified. Here we explore the possibilities of making LNet more resilient, including having it retransmit messages over alternate routes. What can be done in this area is constrained by the design of LNet.

In the following discussion, *node* will often be the shorthand for *local node*, while *peer* will be shorthand for *peer node* or *remote node*.

PUT, ACK, GET, REPLY

Within LNet there are three cases of interest: PUT, PUT+ACK, and GET+REPLY.

- **PUT**: The simplest message type is a bare PUT message. This message is sent on the wire, and after that no further response is expected by LNet. In terms of error handling, this means that a failure to send can result in an error, but any kind of failure after the message has been sent will result in the message being dropped without notification. There is nothing LNet can do in the latter case, it will be up to the higher layers that use LNet to detect what is going on and take whatever corrective action is required.
- **PUT+ACK**: The sender of a Put can ask for an ACK from the receiver. The ACK message is generated by LNet on the receiving node (the peer), and this is done as soon as the Put has been received. In contrast to a bare put this means LNet on the sender can track whether an ACK arrives, and if it does not promptly arrive it can take some action. Such an action can take two forms: inform the upper layers that the ACK took too long to arrive, or retry within LNet itself.
- **GET+REPLY**: With a GET message, there is always a REPLY from the peer. Prior to the GET, the sender and receiver arrange an agreement on the MD that the data for the REPLY must be obtained from, so LNet on the receiving node can generate the REPLY message as soon as the GET has been received. Failure detection and handling is similar to the PUT+ACK case.

The protocols used to implement LNet tend to be connection-oriented, and implement some kind of handshake or ack protocol that tells the sender that a message has been received. As long as the LND actually reports errors to LNet (not a given, alas) this means that in practice the sender of a message can reliably determine whether the message was successfully sent to another node. When the destination node is on the same LNet network, this is sufficient to enable LNet itself to detect failures even in the bare Put case. But in a routed configuration this only guarantees that the LNet router received the message, and if the LNet router then fails to forward it, a bare Put will be lost without trace.

PtlRPC

Users of LNet that send bare Put messages must implement their own methods to detect whether a message was lost. The general rule is simple: the recipient of a Put is supposed react somehow, and if the reaction doesn't happen within a set amount of time, the sender assumes that either the PUT was lost, or the recipient is in some other kind of trouble.

For our purposes PtlRPC is of interest. PtlRPC messages can be classified as Request+Response pairs. Both a Request and a Response are built from one or more Get or PUT messages. A node that sends a PtlRPC Request requires the receiver to send a Response within a set amount of time, and failing this the Request times out and PtlRPC takes corrective action.

Adaptive timeouts add an interesting wrinkle to this mechanism: they allow the recipient of a Request to tell the sender to "please wait", informing it that the recipient is alive and working but not able to send the Response before the normal timeout. For LNet the interesting implication is that while this is going on, there will be some traffic between the sender and recipient. However, this traffic may also be in the form of out-of-band information invisible to LNet.

LNet Interfaces

The interfaces that LNet provides to the upper layers *should* work as follows. Set up an MD (Memory Descriptor) to send data from (for a PUT) or receive data into (for a Get). An event handler is associated with the MD. Then call LNetGet() or LNetPut() as appropriate.

LNetGet()

If all goes well, the event handler sees two events: `LNET_EVENT_SEND` to indicate the GET message was sent, and `LNET_EVENT_REPLY` to indicate the REPLY message was received. Note that the send event can happen after the reply event (this is actually the typical case).

If sending the GET message failed, `LNET_EVENT_SEND` will include an error status, no `LNET_EVENT_REPLY` will happen, and clean up must be done accordingly. If the return value of LNetGet() indicates an error then sending the message certainly failed, but a 0 return does not imply success, only that no failure has yet been encountered.

A damaged REPLY message will be dropped, and does not result in an `LNET_EVENT_REPLY`. Effectively the only way for `LNET_EVENT_REPLY` to have an error status is if LNet detects a timeout before the REPLY is received.

LNetPut() LNET_ACK_REQ

The caller of `LNetPut()` requests an ACK by using `LNET_ACK_REQ` as the value of the `ack` parameter.

A PUT with an ACK is similar to a GET + REPLY pair. The events in this case are `LNET_EVENT_SEND` and `LNET_EVENT_ACK`.

For a PUT, the `LNET_EVENT_SEND` indicates that the MD is no longer used by the LNet code and the caller is free do discard or re-use it.

As with send, `LNET_EVENT_ACK` is expected to only carry an error indication if there was a timeout before the ACK was received.

LNetPut() LNET_NOACK_REQ

The caller of `LNetPut()` requests no ACK by using `LNET_NOACK_REQ` as the value of the `ack` parameter.

A PUT without an ACK will only generate an `LNET_EVENT_SEND`, which indicates that the MD can now be re-used or discarded.

Possible Failures

There are a number of failures we can encounter, only some of which LNet may address.

- Node interface failure: there is some issue with the local interface that prevents it from sending (or receiving) messages.
- Peer interface failure: there is some issue with the remote interface that prevents it from receiving (or sending) messages.
- Path failure: the local interface is working properly, but messages never reach the peer interface.
- Soft peer failure: the peer is properly receiving messages but unresponsive for other reasons.
- Hard peer failure: the peer is down, and unresponsive on all its interfaces.

In a routed LNet configuration these scenarios apply to each hop.

These failures will show up in a number of ways:

- Node interface reports failure. This includes the interface itself being healthy but it noting that the cable connecting it to a switch, or the switch port, is somehow not working right.
- Peer interface not reachable. A peer interface that should be reachable from the node interface cannot be reached. Depending on the error this can result in "fast" error or a timeout in the LND-level protocol.
- Some peer interfaces on a net not reachable. The node interface appears to be OK, but there are interfaces several peers it cannot talk to.
- All peer interfaces on a net not reachable. The node interface appears to be OK, but cannot talk to any peer interface.
- All interfaces of a peer not reachable. All LNDs report errors when talking to a specific peer, but have no problem talking to other peers.
- PUT+ACK or GET+REPLY timeout. The LND gives no failure indication, but the ACK or REPLY takes too long to arrive.
- Dropped PUT. Everything appears to work, except it doesn't.

Let's take a look at what LNet on the node can do in each of these cases.

Node Interface Reports Failure

This is the easiest case to work with. The LND can report such a failure to LNet, and LNet then refrains from using this interface for any traffic.

LNet can mark the interface down, and depending on the capabilities of the LND either recheck periodically or wait for the LND to mark the interface up.

Peer Interface Not Reachable

The peer interface cannot be reached from the node interface, but the node interface can talk to other peers. If the peer interface can be reached from other node interfaces then we're dealing with some path failure. Otherwise the peer interface may be bad. If there is only a single node interface that can talk to the peer interface, then the node cannot distinguish between these cases.

LNet can mark this particular node/peer interface combination as something to be avoided.

When there are paths from more than one node interface to the peer interface, and none of these work, but other peer interfaces do work, then LNet can mark the peer interface as bad. Recovery could be done by periodically probing the peer interface, maybe using LNet Ping as a poor-man's equivalent of an LNet Control Packet.

Some Peer interfaces On A Net Not Reachable

Several peer interfaces on a net cannot be reached from a node interface, but the same node interface can talk to other peers. This is a more severe variant of the previous case.

All Peer Interfaces On A Net Not Reachable

All remote interfaces on a net cannot be reached from a local interface. If there are other, working, interfaces connected to the same net then the balance of probability shifts to the local interface being bad, or there is a severe problem with the fabric.

In practice LNet will not detect "all" remote interfaces being down. But it can detect that for a period of time, no traffic was successfully sent from a local interface, and therefore start avoiding that interface as a whole. Recovery would involve periodically probing the interface, maybe using LNet Ping.

All Interfaces Of A Peer Not Reachable

The node is likely down. There is little LNet can do here, this is a problem to be handled by upper layers. This includes indicating when LNet should attempt to reconnect.

LNet might treat this as the "remote interface not reachable" case for all the interfaces of the remote node. That is, without much difference due to apparently all interfaces of the remote node being down, except for a log message indicating this.

PUT+ACK Or GET+REPLY Timeout

This is the case where the LND does not signal any problem, so the ACK for a PUT or REPLY for a GET should arrive promptly, with the only delays due to credit-based throttling, and yet it does not do so. Note that this assumes that where possible the LND layer already implements reasonably tight timeouts, so that LNet can assume the problem is somewhere else.

LNet will timeout after the configured or passed in transaction timeout and will send an event to the ULP indicating that the PUT/GET has timed out without receiving the expected ACK/REPLY respectively.

Dropped PUT

No problem was signaled by the LND, and there is no ACK that we could time out waiting for. LNet does not have enough information to do anything, so the ULP must do so instead.

If this case must be made tractable, LNet can be changed to make the Ack non-optional.

LNet Resend Handling

When there are multiple paths available for a message, it makes sense to try and resend it on failure. But where should the resending logic be implemented?

The easiest path is to tell upper layers to resend. For example, PtlRPC has some related logic already. Except that when PtlRPC detects a failure, it disconnects, reconnects, and triggers a recovery operation. This is a fairly heavy-weight process, while the type of resending logic desired is to "just try another path" which differs from what exists today and needs to be implemented for each user.

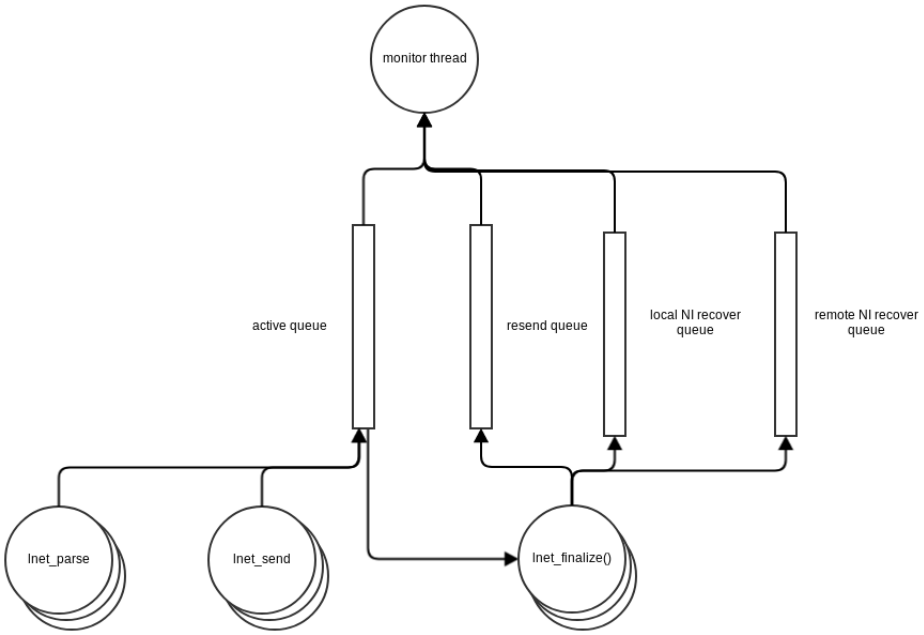
The LNet Resiliency feature shall ensure that failures are delt with at the LNet level by resending the message on the available local and remote interfaces with in a timeout provided.

LNet shall use a trickle down approach for managing timeouts. The ULP (pltrpc or other upper layer protocol) shall provide a timeout value in the call to LNetPut() or LNetGet(). LNet shall use that as the transaction timeout value to wait for an ACK or REPLY. LNet shall further provide a configuration parameter for the number of retries. The number of retries shall allow the user to specify a maximum number of times LNet shall attempt to resend an unsuccessful message. LNet shall then calculate the message timeout by dividing the transaction timeout with the number of retries. LNet shall pass the calculated message timeout to the LND, which will use it to ensure that the LND protocol completes an LNet message within the message timeout. If the LND is not able to complete the message within the provided timeout it will close the connection and drop all messages on that connection. It will afterword proceed to call into LNet via `lnet_finailze()` to notify it of the error encountered.

For PUT that doesn't require an ACK the timeout will be used to provide the transaction timeout to the LND. In that case LNet will resend the PUT if the LND detects an issue with the transmit. LNet shall be able to send a TIMEOUT event to the ULP if the PUT was not successfully transmited. However, if the PUT is successfully transmited, there is no way for LNet to determine if it has been processed properly on the receiving end.

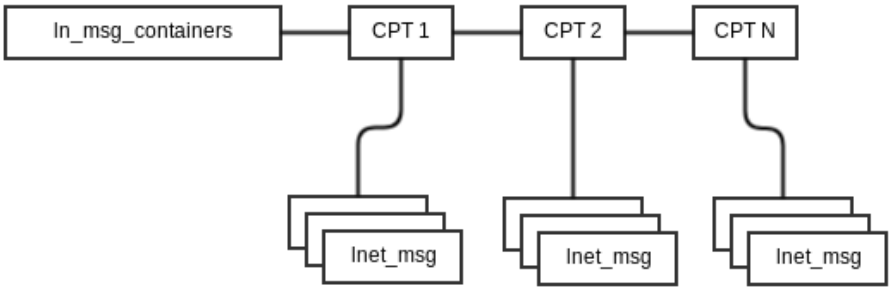
Feature Specification

System Overview



`lnet_msg` is a structure used to keep information on the data that will be transmitted over the wire. It does not itself go over the wire. `lnet_msg` is passed to the LND for transmission.

Before it's passed to the LND it is placed on an active list, `msc_active`. The diagram below describes the datastructures



The CPT is determined by the `lnet_cpt_of_nid_locked()` function. `lnet_send()` running in the context of the calling threads place a message on msc_active just before sending it to the LND. `lnet_parse()` places messages on msc_active when it receives it from the LND.

`msc_active` represent all messages which are currently being processed by LNet.

`lnet_finalize()`, running in the context of the calling threads, likely the LND scheduler threads, will determine if a message needs to be resent and place it on the resend list. The resend list is a list of all messages which are currently awaiting a resend.

A monitor thread monitors and ensures that messages which have expired are finalized. This processing is detailed in later sections.

Resiliency vs. Reliability

There are two concepts that need to stay separate. Reliability of RPC messages and LNet Resiliency. This feature attempts to add LNet Resiliency against local and immediate next hop interface failure. End-to-end reliability is to ensure that upper layer messages, namely RPC messages, are received and processed by the final destination, and take appropriate action in case this does not happen. End-to-end reliability is the responsibility of the application that uses LNet, in this case ptlrpc. Ptlrpc already has a mechanism to ensure this.

To clarify the terminology further, LNET MESSAGE should be used to describe one of the following messages:

- LNET_MSG_PUT
- LNET_MSG_GET
- LNET_MSG_ACK
- LNET_MSG_GET
- LNET_MSG_HELLO

LNET TRANSACTION should be used to describe

- LNET_MSG_PUT, LNET_MSG_ACK sequence
- LNET_MSG_GET, LNET_MSG_REPLY sequence

NEXT-HOP should describe a peer that is exactly one hop away.

The role of LNet is to ensure that an LNET MESSAGE arrives at the NEXT-HOP, and to flag when a transaction fails to complete.

Upper layers should ensure that the transaction it requests to initiate completes successfully, and take appropriate action otherwise.

Failure Areas

There are three areas of failures that LNet needs to deal with:

1. Local Interface failure
2. Remote Interface failure
3. Timeouts
    a. LND detected Timeout
    b. LNet detected Timeout

Timeout values will be provided by the ULP in the LNetPut() and LNetGet() APIs.

Health Value Updates

Two values will be added:

1. health_value: Each NI (local and remote) will have a health value
    a. The health_value will be initialized to 1000
        i. 1000 is chosen in order to granulary select between interfaces based on the value. Otherwise it is arbitrary
    b. When a transient error is detected on an interface, such as a timeout, the health_value is decremented by health_sensitivity
2. health_sensitivity: This is a global configuration parameter. It determines how long an NI takes to recover or how sensitive a system is to message send failure.
    a. An NI's health_value is decremented by health_sensivitiy on a transient error.
    b. An NI is then placed on a queue to recover.
    c. An NI is pinged or pinged from once a second.
    d. Every successful ping would increment the NIs health_value by 1.
    e. It takes health_sensitivity pings to bring the interface back to its original health status.
    f. If a ping fails during the recovery process the health_value is decremented further by health_sensitivity.
        i. This will ensure that an unstable NI which has frequent errors, will be preferred less.
    g. The health_sensitivity can be set to 0 to turn off health evaluation.
        i. That means that an interface will remain healthy no matter what happens.
        ii. Basically turn off NI selection based on health.

Each NI will have a health_value associated with it. Each NI's health value is initialized to 1000

There are two types of errors that could occur on an NI:

1. Hard failures: These are failures communicated by the underlying device driver to the LND and in turn the LND propagates it up to LNet
2. Transient failures: These are failures such as timeouts on the system.

Hard Failures

Hard failures only apply to local interfaces, since there is no way to know if a remote interface has encountered one.

It's possible that the local interface might get into a hard failure scenario by receiving one of these events from the o2iblnd. socklnd needs to be investigated to determine if there are similar cases:

- **IB_EVENT_DEVICE_FATAL**
- **IB_EVENT_PORT_ACTIVE**
- **IB_EVENT_PORT_ERR**
- **RDMA_CM_EVENT_DEVICE_REMOVAL**

In these cases the local interface can not be used any longer. So it can not be selected as part of the selection algorithm. If there are no other interface available, then no messages can be sent out of the node.

A corresponding event can be received to indicate that the interface is operational again.

A new LNet/LND Api will be created to pass these events from the LND to LNet.

Transient failures

Transient Interface failures will be detected in one of two ways

1. Synchronously as a return failure to the call to `lnd_send()`

2. Asynchronously as an event that could be detected at a later point.
    a. These asynchronous events can be a result of a send operations

Synchronous Send Failures

`lnet_select_pathway()` can fail for the following reasons:

1. Shutdown in progress
2. Out of memory
3. Interrup signal received
4. Discovery error.
5. An MD bind failure
    a. `-EINVAL`
    b. `-HOSTUNREACH`
6. Invalid information given
7. Message dropped
8. Aborting message
9. no route found
10. Internal failure

1, 2, 5, 6 and 10 are resource errors and it does not make sense to resend the message as any resend will likely run into the same problem.

Asynchronous Send Failures

LNet should resend the message:

1. On LND transmit timeout
2. On LND connection failure
3. On LND send failure

Resend Handling

When there is a message send failure due to the reasons outlined above. The behavior should be as follows:

1. The local or remote interface health is decremented
2. Failure statistics incremented
3. A resend is issued on a different local interface if there is one available. If there is none available attempt the same interface again.
4. The message will continuously be resent until one of the following criteria is fulfilled:
    a. Message is completed successfully.
    b. Retry-count is reached
    c. Transaction timeout expires

Two new fields will be added to lnet_msg:

1. msg_status - bit field that indicates the type of failure which requires a resend
2. msg_deadline - the deadline for the message calculated by, send time + transaction timeout

```
struct lnet_msg {
...
        __u32 msg_status;
        ktime msg_deadline;
...
}
```

When a message encounters one of the errors above, the LND will update the `msg_status` field appropriately and call `lnet_finalize()`

`lnet_finalize()` will check if the message has timed out or if it needs to be resent and will take action on it. `lnet_finalize()` currently calls `lnet_complete_msg_locked()` to continue the processing. If the message has not been sent, then `lnet_finalize()` should call another function to resend, `lnet_resend_msg_locked()`.

`lnet_resend_msg_locked()`shall queue the message on a resend queue and wake up a thread responsible for resending messages, the monitor thread portrayed in the above diagram.

When a message is initially sent it's tagged with a deadline for this message. The message will be placed on the active queue. If the message is not completed within that timeout it will be finalized and removed from the active queue. A timeout event will be passed to the ULP.

If the LND times out and LNet attemps to resend, it'll place the message on the resend queue. A message can be on both the active and resend queue.

As shown in the diagram above both `lnet_send()` and `lnet_parse()` put messages on the active queue. `lnet_finalize()` consumes messages off the active queue when it's time to decommit them.

When the LND calls `lnet_finalize()` on a timed out message, `lnet_finalize()` will put the message on the resend queue and wake up the monitor thread.

The Monitor Thread

The router checker thread will be refactored to full fill the following responsibilities:

1. Check the active queue once per second for expired messages
    a. The monitor thread will wake up ever second and check the top of the active queue, IE the oldest message on the list. If that message has expired it updates its status to TIMEDOUT and finalizes it. Finalizeing the message will include removing it from the active queue and the resend queue. It then moves on to the next message on the list and stops once it find a message that has not expired.
2. Check if there are any messages to resend on the resend_queue
    a. If there are any messages queued, it'll call `lnet_send()` on each one.
3. Check if there are any peers on the local_ni recovery queue.
    a. local_nis are a bit tricky to recover. How do you determine if a local NI is good again. Do we ping a random peer NI on the same network as the local NI? If so then what if this local NI has a problem? We could be introducing other failure handling not associated with the local NI recovery during its recovery process.
    b. Best approach at this time is to ping itself.
        i. Pinging itself will force the ping message to travel down the entire stack, LND, Verbs/TCP and IB/HFI/IP. This should be sufficient to determine if the interface has recovered from the transient error encountered.
        ii. The time delay to recover the interface will also allow for the LNDs queue to empty out under congestion.
4. Check if there are any peers on the remote_ni recovery queue
    a. ping the remote ni
    b. Unfortunately, that could result in using an unhealty local NI, but there is no way around that.
        i. In that case we will manage the health_value of the local NI and remote NI as described above.

The assumption is that under normal circumstances the number of re-sends should be low, so the thread will not add any logic to pace out the resend rate, such as what `lnet_finalize()` does.

In case of immediate failures, for example route failure, the message will not make it on the network. There is a risk that immediate failure could trigger a burst of resends for the message. This could be exaggerated if there is only one interface in the system.

This will be metigated by having a maximum number of retry count. This is a configured value and will cap the number of resends in this case.

Setting retry count to 0 will turn off retries completely and will trigger a message to fail and propagated up on first failure encountered.

It is possible that a message can be on the resend queue when it either completes or times out. In both of these case it will be removed from the resend queue as well as the active queue and finalized.

Remote triggered timeout

A PUT not receiving an ACK or a GET not receiving a REPLY is considered a remote timeout. This could happen if the peer is too busy to respond in a timely way or if the peer is going through a reboot, or other unforseen circustances.

The other case which falls into this category is LND timeouts because of missing LND acknowledgment, ex IBLND_MSG_PUT_ACK.

In these cases LNet can not issue a resend safely. The message could've already been received on the remote end and processed. In this case we must add the ability to handle duplicate requests, which is outside the scope of this project.

Therefore, handling this category of failures will be delegated to the ULP. LNet will pass up a timeout event to the ULP.

Protection

The message will continue to be protected by the LNet net CPT lock to ensure mutual access.

When the message is committed, lnet_msg_commit(), the message cpt is assigned. This cpt value is then used to protect the message in subsequent usages. Relevant to this discussion is when the message is examined in `lnet_finalize()` and in the monitor thread and either removed from the active queue or placed on the resend queue.

Timeout Value

This section discusses how LNet shall calculate its timeouts

There are two options:

1. ULP provided timeout
2. LNet configurable timeout

ULP Provided Timeout

The ULP, ex: ptlrpc, will set the timeout in the lnet_libmd structure. This timeout indicates how long the ptlrpc is willing to wait for an RPC response.

LNet can then set its own timeouts based on that timeout.

The draw back of this approach is that the timeouts set is specific to the ULP. For example in ptlrpc this would be the time to wait for an RPC reply.

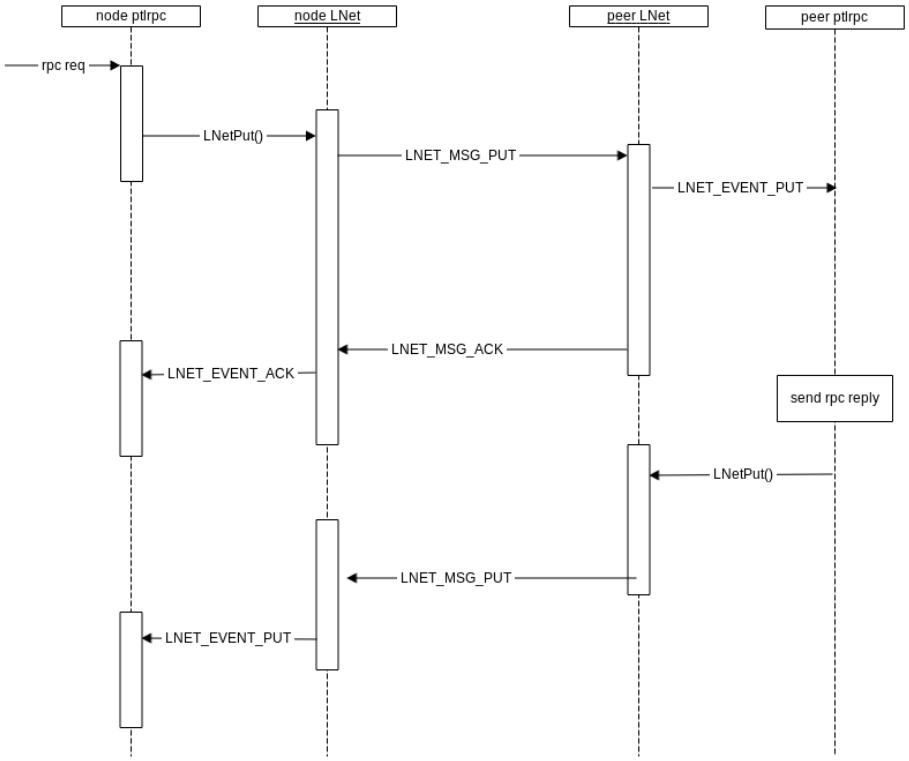
As shown in the diagram below the timeout provided by ptlrpc would cover the RPC response sent by the peer ptlrpc. While LNet needs to ensure that a single LNET_MSG_PUT makes it to peer LNet.

Therefore, the ULP timeout is related to the LNet transaction timeout only in the sense that it must be less than ULP timeout, but a reasonable LNet transaction timeout can not be derived from the ULP timeout, other than to say it must be less.

This might be enough of a reason to pass the ULP down and have the following logic:

```
transaction_timeout = min(ULP_timeout / num_timeout, global_transaction_timeout / num_retries)
```

This way the LNet transaction timeout can be set a lot less than the ULP timeout, considering that the ULP timeout can vary its timeout, based on an adaptive backoff algorithm.



This trickle down approach has the advantage of simplifying the configuration of the LNet Resiliency feature, as well as making the timeout consistent through out the system, instead of configuring the LND timeout to be much larger than the pltrpc timeout as it is now.

The timeout parameter will be passed down to LNet by setting it in the `struct lnet_libmd`

```
struct lnet_libmd {
        struct list_head md_list;
        struct lnet_libhandle md_lh;
        struct lnet_me *md_me;
        char *md_start;
        unsigned int md_offset;
        unsigned int md_length;
        unsigned int md_max_size;
        int md_threshold;
        int md_refcount;
        unsigned int md_options;
        unsigned int md_flags;
        unsigned int md_niov; /* # frags at end of struct */
        void *md_user_ptr;
        struct lnet_eq *md_eq;
        struct lnet_handle_md md_bulk_handle;
    /*
         * timeout to wait for this MD completion
         */
    ktime md_timeout;
        union {
                struct kvec»····iov[LNET_MAX_IOV];
                lnet_kiov_t»····kiov[LNET_MAX_IOV];
        } md_iov;
};
```

LNet Configurable timeout

A transaction timeout value will be configured in LNet and used as described above. This approach avoids passing down the ULP timeout and relies on the sys admin to set the timeout in LNet to a reasonable value, based on the network requirements of the system.

That timeout value will still be used to calculate the LND timeout as in the above approach.

Conclusion on Approach

The second approach will be taken for the first phase, as it reduces the scope of the work and allows the project to meet the deadline for 2.12.

Selection Algorithm

The selection algorithm will be modified to take health into account and will operate according to the following logic:

```
for every peer_net in peer {
        local_net = peer_net

        if peer_net is not local
                select a router
                local_net = router->net

        for every local_ni on local_net
                check if local_ni has best health_value
                check if local_ni is nearest MD NUMA
                check if local_ni has the most available credits
                check if we need to use round robin selection
                If above criteria is satisfied
                        best_ni = local_ni

        for every peer_ni on best_ni->net
                check if peer_ni has best health value
                check if peer_ni has the most available credits
                check if we need to use round robin selection
                If above criteria is satisfied
                        best_peer_ni = peer_ni

        send(best_ni, peer_ni)
}
```

The above algorithm will always prefer NI's that are the most healthy. This is important because dropping even one message will likely result in client evictions. So it is important to always ensure we're using the best path possible.

LND Interface

LNet shall calculate the message timeout as follows:

message timeout = transaction timeout / retry count

The message timeout will be stored in the lnet_msg structure and passed down to the LND via lnd_send().

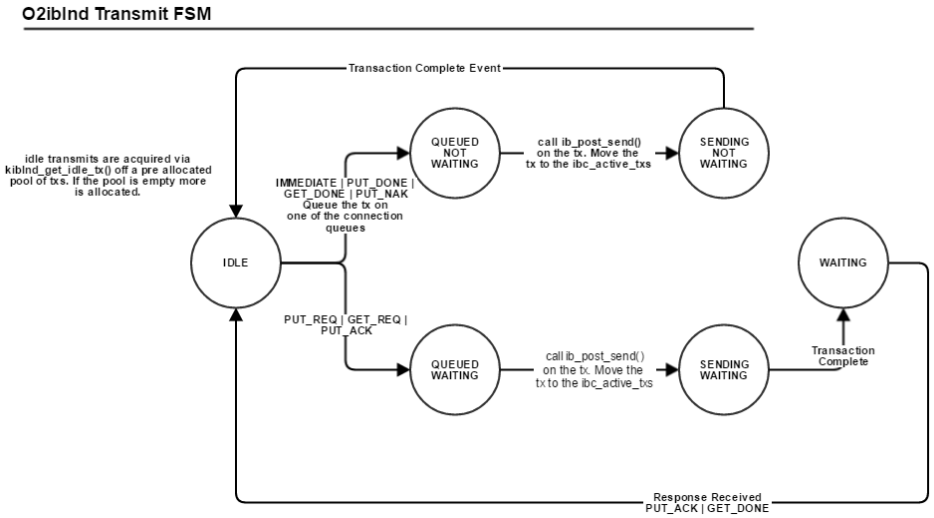
LND Transmits (o2iblnd specific discussion)

ULP requests from LNet to send a GET or a PUT via `LNetGet()` and `LNetPut()` APIs. LNet then calls into the LND to complete the operation. The LND can complete the LNet PUT/GET via a set of LND messages as shown in the diagrams below.

When the LND transmits the LND message it sets a `tx_deadline` for that particular transmit. This `tx_deadline` remains active until the remote has confirmed receipt of the message, if an aknwoledgment is expected or if a no acknowledgement is expected then when the tx is completed the tx_deadline is completed. Receipt of the message at the remote is when LNet is informed that a message has been received by the LND, done via `lnet_parse()`, then LNet calls back into the LND layer to receive the message.

By handling the `tx_deadline` properly we are able to account for almost all next-hop failures. LNet would've done its best to ensure that a message has arrived at the immediate next hop.

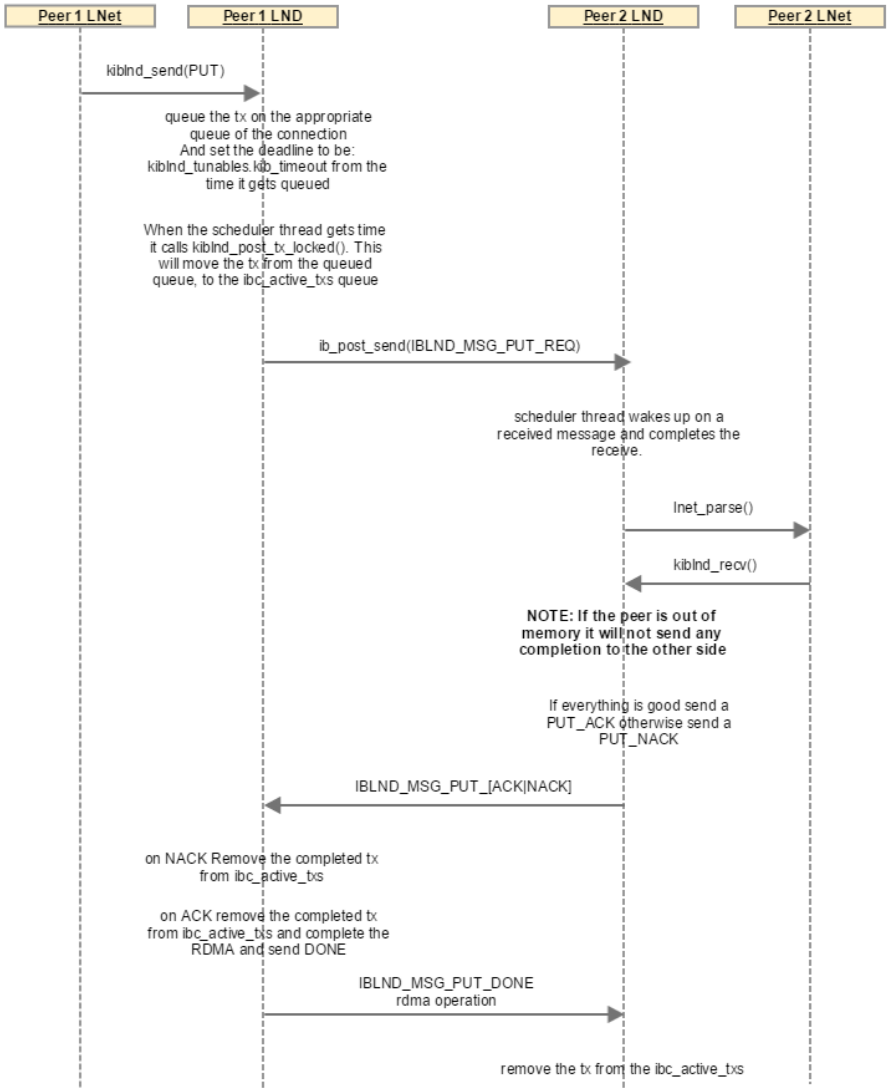
The `tx_deadline` is LND-specific, and derived from the `timeout` (or `sock_timeout`) module parameter of the LND.



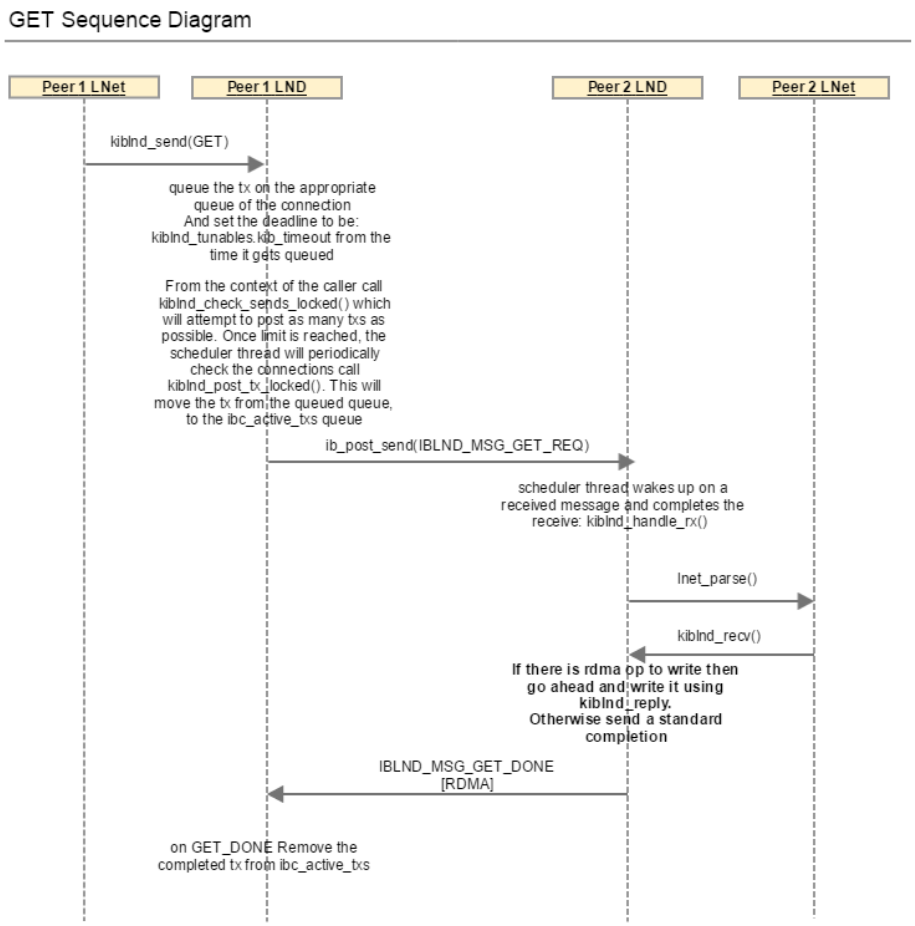
LND timeout

PUT

PUT Sequence Diagram



GET



A third type of message that the LND sends is the IBLND_MSG_IMMEDIATE. The data is embedded in the message and posted. There is no handshake in this case.

For the PUT case described in the sequence diagram, the initiator sends two messages:

1. IBLND_MSG_PUT_REQ
2. IBLND_MSG_PUT_DONE

Both of these messages are sent using the same tx structure. The tx is allocated and placed on a waiting queue. When the IBLND_MSG_PUT_ACK is received the waiting tx is looked up and used to send the IBLND_MSG_PUT_DONE.

When `kiblnd_queue_tx_locked()` is called for IBLND_MSG_PUT_REQ it sets the tx_deadline as follows:

```
timeout_ns = *kiblnd_tunables.kib_timeout * NSEC_PER_SEC;
tx->tx_deadline = ktime_add_ns(ktime_get(), timeout_ns);
```

When kiblnd_queu_tx_locked() is called for IBLND_MSG_PUT_DONE it reset the tx_deadline again.

This presents an obstacle for the LNet Resiliency feature. LNet provides a timeout for the LND as described above. From LNet's perspective this deadline is for the LNet PUT message. However, if we simply use that value for the timeout_ns calculation, then in essence will will be waing for 2 * LND timeout for the completion of the LNet PUT message. This will mean less re-transmits.

Therefore, the LND, since it has knowledge of its own protocols will need to divide the timeout provided by LNet by the number of transmits it needs to do to complete the LNet level message:

1. LNET_MSG_GET: Requires only IBLND_MSG_GET_REQ. Use the LNet provided timeout as is.
2. LNET_MSG_PUT: Requires IBLND_MSG_PUT and IBLND_MSG_PUT_DONE to complete the LNET_MSG_PUT. Use LNet provided timeout / 2
3. LNET_MSG_GET/LNET_MSG_PUT with < 4K payload: Requires IBLND_MSG_IMMEDIATE. Use LNet provided timeout as is.

System Timeouts

There are multiple timeouts kept at different layers of the code. The LNet Resiliency will attempt to reduce the complexity and ambiguity of setting the timeouts in the system.

This will be done by using a trickle down approach as mentioned before. The top level transaction timeout will be provided to LNet for each PUT/GET send request. If one is not provided LNet will use a configurable default.

LNet will calculate the following timeouts from the transaction timeout:

1. Message timeout = Transaction timeout / retry count
2. LND timeout = Message timeout / number of LND messages used to complete an LNet PUT/GET

Implementation Specifics

Reasons for timeout

The discussion here refers to the LND Transmit timeout.

Timeouts could occur due to several reasons:

1. The message is on the sender's queue and is not posted within the timeout
    a. This indicates that the local interface is too busy and is unable to process the messages on its queue.
2. The message is posted but the transmit is never completed
    a. An actual culprit can not be determined in this scenario. It could be a sender issue, a receiver issue or a network issue.
3. The message is posted, the transmit is completed, but the remote never acknowledges.
    a. In the IBLND, there are explicit acknowledgements in most cases when the message is received and forwarded to the LNet layer. Look below for more details.
    b. If an LND message is in waiting state and it didn't receive the expected response, then this indicates an issue at the remote's LND, either at the lower protocol, IB/TCP, or the notification at the LNet layer is not being processed in a timely fashion.

Each of these scenarios can be handled differently

Desired Behavior

The desired behavior is listed for each of the above scenarios:

Scenario 1 - Message not posted

1. Connection is closed
2. The local interface health is decremented
3. Failure statistics incremented
4. A resend is issued.
5. Selection algorithm will prefer less the unhealthy NI

Scenario 2 - Transmit not completed

1. Connection is closed
2. The local and remote interface health is updated
3. Failure statistics incremented on both local and remote
4. Selection algorithm will prefer less the unhealthy NIs

Scenario 3 - No acknowledgement by remote

1. Connection is closed
2. The remote interface health is updated
3. Failure statistics incremented
4. Selection algorithm will prefer less the unhealthy NIs

Selection Algorithm with Health

Algorithm Parameters

Parameter	Values	
SRC NID	Specified (A)	Not specified (B)
DST NID	local (1)	not local (2)
DST NID	MR (C)	NMR (D)

Note that when communicating with an NMR peer we need to ensure that the source NI is always the same: there are a few places where the upper layers use the src nid from the message header to determine its originating node, as opposed to using something like a UUID embedded in the message. This means when sending to an NMR node we need to pick a NI and then stick with that going forward.

Note: When sending to a router that scenario boils down to considering the router as the next-hop peer. The final destination peer NIs are no longer considered in the selection. The next-hop can then be MR or non-MR and the code will deal with it accordingly.

A1C - src specified, local dst, mr dst

- find the local ni given src_nid
  - if no local ni found fail
  - if local ni found is down, then fail
- find peer identified by the dst_nid
- select the best peer_ni for that peer
  - take into account the health of the peer_ni (if we just demerit the peer_ni it can still be the best of the bunch. So we need to keep track of the peer_nis/local_nis a message was sent over, so we don't revisit the same ones again. This should be part of the message)
  - If this is a resend and the resend peer_ni is specified, then select this peer_ni if it is healthy, otherwise continue with the algorithm.
  - if this is a resend, do not select the same peer_ni again unless no other peer_nis are available and that peer_ni is not in a HARD_ERROR state.

A2C - src specified, route to dst, mr dst

- find local ni given src_nid
  - if no local ni found fail
  - if local ni found is down, then fail
- find router to dst_nid
  - If no router present then fail.
- find best peer_ni (for the router) to send to
  - take into account the health of the peer_ni
  - If this is a resend and the resend peer_ni is specified, then select this peer_ni if it is healthy, otherwise continue with the algorithm.
  - If this is a resend and the peer_nis is not specified, do not select the same peer_ni again. The original destination NID can be found in the message.
    - Keep trying to send to the peer_ni even if it has been used before, as long as it is not in a HARD_ERROR state.

A1D - src specified, local dst, nmr dst

- find local ni given src nid
  - if no local_ni found fail
  - if local ni found is down, then fail
- find peer_ni using dst_nid
- send to that peer_ni
  - If this is a resend retry the send on the peer_ni unless that peer_ni is in a HARD_ERROR state, then fail.

A2D - src specified, route to dst, nmr dst

- find local_ni given the src_nid
  - if no local_ni found fail
  - if local ni found is down, then fail
- find router to go through to that peer_ni
- send to the NID of that router.
  - If this is a resend retry the send on the peer_ni unless that peer_ni is in a HARD_ERROR state, then fail.

B1C - src any, local dst, mr dst

- select the best_ni to send from, by going through all the local_nis that can reach any of the networks the peer is on
  - consider local_ni health in the selection by selecting the local_ni with the best health value.
  - If this is a resend do not select a local_ni that has already been used.
- select the best_peer_ni that can be reached by the best_ni selected in the previous step
  - If this is a resend and the resend peer_ni is specified, then select this peer_ni if it is healthy, otherwise continue with the algorithm.
  - If this is a resend and the resend peer_ni is not specified do not consider a peer_ni that has already been used for sending as long as there are other peer_nis available for selection. Loop around and re-use peer-nis in round robin.
    - peer_nis that are selected cannot be in HARD_ERROR state.
- send the message over that path.

B2C - src any, route to dst, mr dst

- find the router that can reach the dst_nid
- find the peer for that router. (The peer is MR)
- go to B1C

B1D - src any, local dst, nmr dst

- find `peer_ni` using `dst_nid` (non-MR, so this is the only `peer_ni` candidate)
  - no issue if `peer_ni` is healthy
  - try this `peer_ni` even if it is unhealthy if this is the 1st attempt to send this message
  - fail if resending to an unhealthy `peer_ni`
- pick the preferred local_NI for this `peer_ni` if set

    - If the preferred local_NI is not healthy, fail sending the message and let the upper layers deal with recovery.
    - otherwise if preferred local_NI is not set, then pick a healthy local NI and make it the preferred NI for this `peer_ni`
- send over this path

B2D - src any, route dst, nmr dst

- find route to `dst_nid`
- find `peer_ni` of router
    - no issue if `peer_ni` is healthy
    - try this `peer_ni` even if it is unhealthy if this is the 1st attempt to send this message
    - fail if resending to an unhealthy `peer_ni`
- pick the preferred local_NI for the `dst_nid` if set
    - If the preferred local_NI is not healthy, fail sending the message and let the upper layers deal with recovery.
    - otherwise if preferred local_NI is not set, then pick a healthy local NI and make it the preferred NI for this `peer_ni`
- send over this path

Work Items

- refactor lnet_select_pathway() as described above.
- Health Value Maintenance/Demerit system
- Selection based on Health Value and not resending over already used interfaces unless non are available.
- Handling the new events in IBLND and passing them to LNet
- Handling the new events in SOCKLND and passing them to LNet
- Adding LNet level transaction timeout (or reuse the peer timeout) and cancelling a resend on timeout
- Handling timeout case in ptlrpc

Progress

```
LNet Health
        Refactor lnet_select_pathway() - DONE
        add health value per ni - DONE
        add lnet_health_range - DONE
        handle local timeouts - DONE
                When re-sending a message we don't need to ensure we send to the same peer_ni as the original
send. There are two cases to consider:
                        MR peer: we can just use the current selection algorithm to resend a message
                        Non-MR peer: there will only be on peer_ni anyway (or preferred NI will be set) and
we'll need to use the same local NI when sending to a Non-MR.
        Modify the LNDs to set the appropriate error code on timeout
                handle tx timeout due being stuck on the queues for too long
                        Due to local problem.
                At this point we should be able to handle trying different interfaces if there is an interface
timeout
                o2iblnd
                socklnd
        Introduce retry_count
                Only resend up to the retry_count
                This should be user configurable
                Should have a max value of 5 retries
        Rate limit resend rate
                Introduce resend_interval
                        Make sure to pace out the resends by that interval
                We need to guard against situations where there is an immediate failure which triggers an
immediate resend, causing a resend tight loop
        Refactor the router pinger thread to handle resending.
                lnet_finalize() queues those messages on a queue and wakes up the router pinger thread
                router pinger wakes up every second (or if woken up manually) goes through the queue, timesout
and fails any messages that have passed their deadline. Checks if a message to be resent is not being resent
before its resend interval. Resends any messages that need to be resent.
        Introduce an LND API to read the retransmit timeout.
                Calculate the message timeout as follows:
                        message timeout = (retry count * LND transmit timeout) + (resend interval * retry count)
                        Message timeout is the timeout by which LNet abandons retransmits
                                This implies that LNet has detected some sort of a failure while sending a
message
                use the message timeout instead of the peer timeout as the deadline for the message
                If the message timesout a failure event is propagated to the top layer.
                o2iblnd
```

```
                socklnd
        handle local NIs down events from the LND.
                NIs are flagged as down and are not considered as part of the selection process.
                Can only come up by another event from the LND.
                o2iblnd
                socklnd
        Move the peer timeout from the LND to the LNet.
                It should still be per NI.
        Add userspace support for setting retry count
        Add userspace support for setting retransmit interval
        Add peer_ni_healthvalue
                This value will reflect the health of the peer_ni and should be initially set the peer credits.
        Modify the selection algorithm to select the peer_ni based on the average of the health value and the
credits
        Adjust the peer_ni health value due to failure/successs
                On Success the health value should be incremented if it's not at its maximum value.
                On Failure the health value should be decremented (stays >= 0)
                        Failures will either be due to remote tx timeout or network error
        Modify the LNDs to set the appropriate error code on tx timeout
                o2iblnd
                socklnd
        Handle transaction timeout
                Transaction timeout is the deadline by which LNet knows that a PUT or a GET did not receive the
ACK or REPLY respectively.
                When a PUT or a GET is sent successfully.
                It is then put on a queue if it expects and ACK or a REPLY
                router pinger will wake up every second and will check if these messages have not received the
expected response within the timeout specified. If not then we'll need to time it out.
        Provide a mechanism to over ride the transaction timeout.
                When sending a message the caller of LNetGet()/LNetPut() should specify a timeout for the
transaction. If not provided then it defaults to the global transaction timeout.
        Add a transaction timeout even to be send to the upper layer.
        Handle transaction timeout in the upper layer (ptlrpc)
        Add userspace support for maximum transaction timeout
                This was added in 2.11 to solve the blocked mount
        Add the following statistics
                The number of resends due to local tx timeout per local NI
                The number of resends due to the remote tx timeout per peer NI
                The number of resends due to a network timeout per local and peer NI
                The number of local tx timeouts
                The number of remote tx timeouts
                The number of network timeouts
                The number of local interface down events
                The number of local interface up events.
                The average time it takes to successfully send a message per peer NI
                The average time it takes to successfully complete a transaction per peer NI
```

O2IBLND Detailed Discussion

Overview

There are two types of events to account for:

1. Events on the RDMA device itself
2. Events on the cm_id

Both events should be monitored because they provide information on the health of the device and connection respectively.

ib_register_event_handler() can be used to register a handler to handle events of type 1.

a cm_callback can be register with the cm_id to handle RMDA_CM events.

There is a group of events which indicate a fatal error

RDMA Device Events

Below are the events that could occur on the RDMA device. Highlighted in **BOLD RED** are the events that should be handled for health purposes.

- IB_EVENT_CQ_ERR
- IB_EVENT_QP_FATAL
- IB_EVENT_QP_REQ_ERR
- IB_EVENT_QP_ACCESS_ERR
- IB_EVENT_COMM_EST
- IB_EVENT_SQ_DRAINED
- IB_EVENT_PATH_MIG
- IB_EVENT_PATH_MIG_ERR
- **IB_EVENT_DEVICE_FATAL**
- **IB_EVENT_PORT_ACTIVE**
- **IB_EVENT_PORT_ERR**
- IB_EVENT_LID_CHANGE
- IB_EVENT_PKEY_CHANGE
- IB_EVENT_SM_CHANGE
- IB_EVENT_SRQ_ERR
- IB_EVENT_SRQ_LIMIT_REACHED
- IB_EVENT_QP_LAST_WQE_REACHED
- IB_EVENT_CLIENT_REREGISTER
- IB_EVENT_GID_CHANGE

Communication Events

Below are the events that could occur on a connection. Highlighted in BOLD RED are the events that should be handled for health purposes.

- RDMA_CM_EVENT_ADDR_RESOLVED: Address resolution (rdma_resolve_addr) completed successfully.
- RDMA_CM_EVENT_ADDR_ERROR: Address resolution (rdma_resolve_addr) failed.
- RDMA_CM_EVENT_ROUTE_RESOLVED: Route resolution (rdma_resolve_route) completed successfully.
- RDMA_CM_EVENT_ROUTE_ERROR: Route resolution (rdma_resolve_route) failed.
- RDMA_CM_EVENT_CONNECT_REQUEST: Generated on the passive side to notify the user of a new connection request.
- RDMA_CM_EVENT_CONNECT_RESPONSE: Generated on the active side to notify the user of a successful response to a connection request. It is only generated on rdma_cm_id's that do not have a QP associated with them.
- RDMA_CM_EVENT_CONNECT_ERROR: Indicates that an error has occurred trying to establish or a connection. May be generated on the active or passive side of a connection.
- **RDMA_CM_EVENT_UNREACHABLE**: Generated on the active side to notify the user that the remote server is not reachable or unable to respond to a connection request. If this event is generated in response to a UD QP resolution request over InfiniBand, the event status field will contain an errno, if negative, or the status result carried in the IB CM SIDR REP message.
- RDMA_CM_EVENT_REJECTED: Indicates that a connection request or response was rejected by the remote end point. The event status field will contain the transport specific reject reason if available. Under InfiniBand, this is the reject reason carried in the IB CM REJ message.
- RDMA_CM_EVENT_ESTABLISHED: Indicates that a connection has been established with the remote end point.
- RDMA_CM_EVENT_DISCONNECTED: The connection has been disconnected.
- **RDMA_CM_EVENT_DEVICE_REMOVAL**: The local RDMA device associated with the rdma_cm_id has been removed. Upon receiving this event, the user must destroy the related rdma_cm_id.
- RDMA_CM_EVENT_MULTICAST_JOIN: The multicast join operation (rdma_join_multicast) completed successfully.
- RDMA_CM_EVENT_MULTICAST_ERROR: An error either occurred joining a multicast group, or, if the group had already been joined, on an existing group. The specified multicast group is no longer accessible and should be rejoined, if desired.
- RDMA_CM_EVENT_ADDR_CHANGE: The network device associated with this ID through address resolution changed its HW address, eg following of bonding failover. This event can serve as a hint for applications who want the links used for their RDMA sessions to align with the network stack.
- RDMA_CM_EVENT_TIMEWAIT_EXIT: The QP associated with a connection has exited its timewait state and is now ready to be re-used. After a QP has been disconnected, it is maintained in a timewait state to allow any in flight packets to exit the network. After the timewait state has completed, the rdma_cm will report this event.

Health Handling

Handling Asynchronous Events

- A callback mechanism should be provided by LNet to the LND to report failure events
  Some translation matrix from LND specific errors to LNet specific errors should be created
  - Each LND would create the mapping
- Whenever an event occurs the indicates a fatal error on the device the LNet callback should be called.
- LNet should transition the local NI or remote NI appropriately and take measures to close the connections on that specific device.

Handling Errors on Sends

If a request to send a message ends in an error: Example a connection error (as seen with the wrong device responding to ARP), then LNet should pick another local device to send from.

- There are a class of errors which indicate a problem in Local NI
- RDMA_CM_EVENT_DEVICE_REMOVAL - This device is no longer present. Should never be used.
- There are a class of errors which indicate a problem in remote NI
- RDMA_CM_EVENT_ADDR_ERROR - The remote address is errnoeous. Should not be used.
- RDMA_CM_EVENT_ADDR_RESOLVED with an error. The remote address can not be resolved
- RDMA_CM_EVENT_ROUTE_ERROR - No route to remote address. Should result in the peer_ni not to be used. But a retry can be done a bit later via time.
- RDMA_CM_EVENT_UNREACHABLE - Remote side is unreachable. Retry after a while.
- RDMA_CM_EVENT_CONNECT_ERROR - problem with connection. Retry after a while.

- RDMA_CM_EVENT_REJECTED - Remote side is rejecting connection. Retry after a while.
- RDMA_CM_EVENT_DISCONNECTED - Move outstanding operations to a different pair if available.

Handling Timeout

This is probably the trickiest situation. Timeout could occur because of network congestion, or because the remote side is too busy, or because it's dead, or hung, etc.

Timeouts are being kept in the LND (o2iblnd) on the transmits. Every transmit which is queued is assigned a deadline. If it expires then the connection on which this transmit is queued, is closed.

peer_timout can be set in routed and non-routed scenario, which provides information on the peer.

Timeouts are also being kept at ptlrpc. These are rpc timeouts.

Refer to section 32.5 in the manual for a description of how RPC timeouts work.

Also refer to section 27.3.7 for LNet Peer Health information.

Given the presence of various timeouts, adding yet another timeout on the message, will further complicate the configuration, and possibly cause further hard to debug issues.

One option to consider is to use the peer_timout feature to recognize when peer_nis are down, and update the peer_ni health information via this mechanism. And let the LND and RPC timeouts take care of further resends.

High Level Design

Callback Mechanism

[Olaf: bear in mind that currently the LND already reports status to LNet through `lnet_finalize()`]

Add an LNet API:
- lnet_report_lnd_error(lnet_error_type type, int errno, struct lnet_ni *ni, struct lnet_peer_ni *lpni)
- LND will map its internal errors to lnet_error_type enum.

```
enum lnet_error_type {
        LNET_LOCAL_NI_DOWN, /* don't use this NI until you get an UP */
        LNET_LOCAL_NI_UP, /* start using this NI */
        LNET_LOCAL_NI_SEND_TIMOUT, /* demerit this NI so it's not selected immediately, provided there are
other healthy interfaces */
        LNET_PEER_NI_ADDR_ERROR, /* The address for the peer_ni is wrong. Don't use this peer_NI */
        LNET_PEER_NI_UNREACHABLE, /* temporarily don't use the peer NI */
        LNET_PEER_NI_CONNECT_ERROR, /* temporarily don't use the peer NI */
        LNET_PEER_NI_CONNECTION_REJECTED /* temporarily don't use the peer NI */
};
```

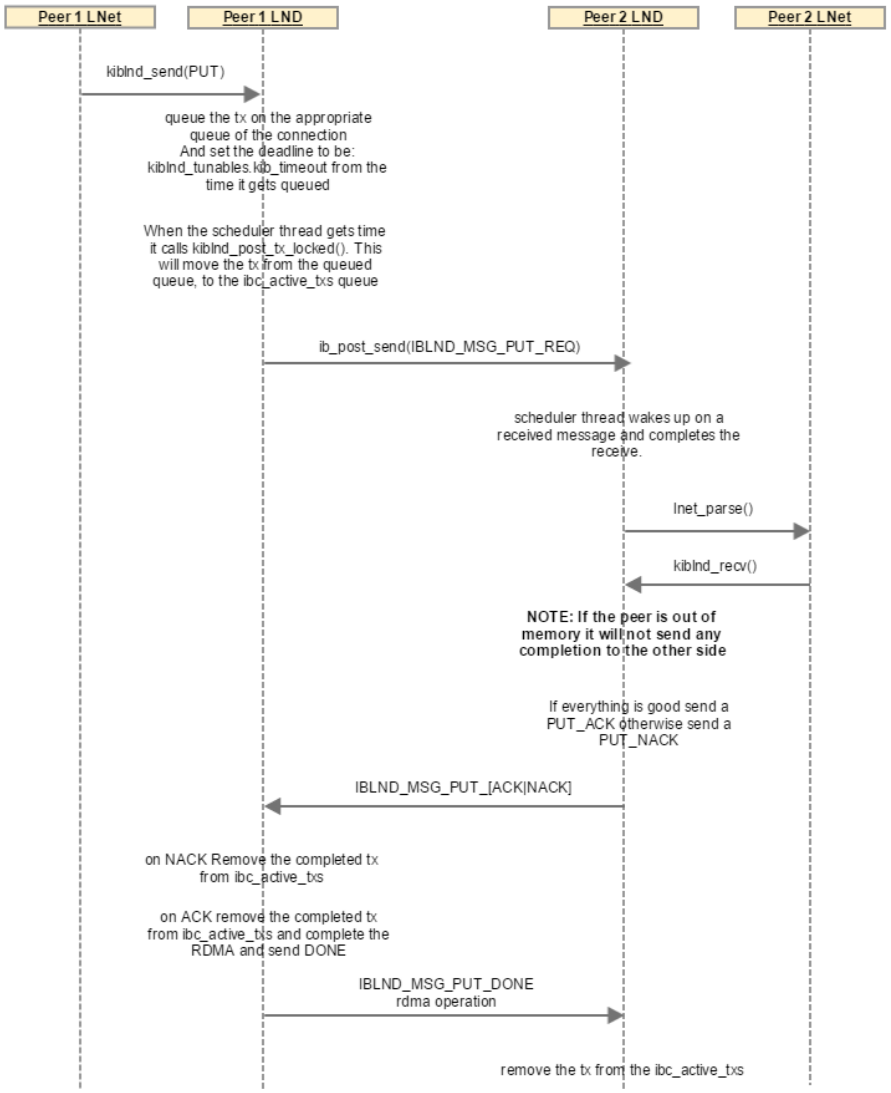
- Although some of the actions LNet will take is the same for different errors, it's still a good idea to keep them separate for statistics and logging.
- on LNET_LOCAL_NI_DOWN set the ni_state to STATE_FAILED. In the selection algorithm this NI will not be picked.
- on LNET_LOCAL_NI_UP set the ni_state to STATE_ACTIVE. In the selection algorithm this NI will be selected.
- Add a state in the peer_ni. This will indicate if it usable or not.
- on LNET_PEER_NI_ADDR_ERROR set the peer_ni state to FAILED. This peer_ni will not be selected in the selection algorithm.
- Add a health value (int). 0 means it's healthy and available for selection.
- on any LNet_PEER_NI_[UNREACHABLE | CONNECT_ERROR | CONNECT_REJECTED] decrement this value.
- That value indicates how long before we use it again.
- A time before use in jiffies is stored. The next time we examine this peer_NI for selection, we take a look at that time. If it has been passed we select it, but we do not increment this value. The value is set to 0 only if there is a successful send to this peer_ni.
- The net effect is that if we have a bad peer_ni, the health value will keep getting decremented, which will mean it'll take progressively longer to reuse it.
- This algorithm is in effect only if there are multiple interfaces, and some of them are healthy. If none of them are healthy (IE the health value is negative), then select the least unhealthy peer_ni (the one with greatest health value).
- The same algorithm can be used for local NI selection

Timeout Handling

LND TX Timeout

PUT

PUT Sequence Diagram



As shown in the above diagram whenever a tx is queued to be sent or is posted but haven't received confirmation yet, the tx_deadline is still active. The scheduler thread checks the active connections for any transmits which has passed their deadline, and then it closes those connections and notifies LNet via lnet_notify().

The tx timeout is cancelled when in the call kiblnd_tx_done(). This function checks 3 flags: tx_sending, tx_waiting and tx_queued. If all of them are 0 then the tx is closed as completed. The key flag to note is tx_waiting. That flag indicates that the tx is waiting for a reply. It is set to 1 in kiblnd_send, when sending the PUT_REQ or GET_REQ. It is also set when sending the PUT_ACK. All of these messages expect a reply back. When the expected reply is received then tx_waiting is set to 0 and kiblnd_tx_done() is called, which eventually cancels the tx_timeout, by basically removing the tx from the queues being checked for the timeout.

The notification in the LNet layer that the connection has been closed can be used by MR to attempt to resend the message on a different peer_ni.

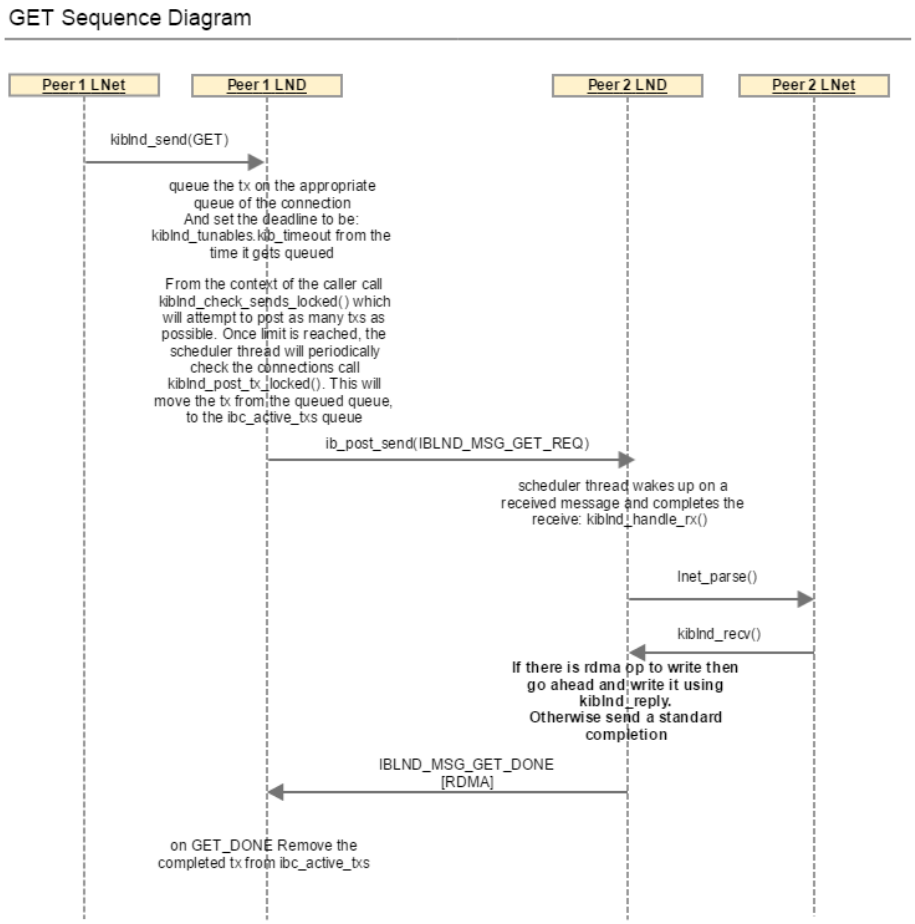
<TBD: I don't think that LND attempts to automatically reconnect to the peer if the connection gets torn down because of a tx_timeout.>

TX timeout is exactly what we need to determine if the message has been transmitted successfully to the remote side. If it has not been transmitted successfully we can attempt to resend it on different peer_nis until we're either successful or we've exhausted all of the peer_nis.

The reason for the TX timeout is also important:

1. TX timeout can be triggered because the TX remains on one of the outgoing queues for too long. This would indicate that there is something wrong with the local NI. It's either too congested or otherwise problematic. This should result in us trying to resend using a different local NI but possibly to the same peer_ni.
2. TX timeout can be triggered because the TX is posted via (ib_post_send()) but it has not completed. In this case we can safely conclude that the peer_ni is either congested or otherwise down. This should result in us trying to resent to a different peer_ni, but potentially using the same local NI.

GET



After the completion of an o2iblnd tx ib_post_send(), a completion event is added to the completion queue. This triggers kiblnd_complete to be called. If this is an IBLND_WID_TX then kiblnd_tx_complete() is called, which calls kiblnd_tx_done() if the tx is not sending, waiting or queued. In this case the tx_timeout is closed.

In summary, the tx_timeout serves to ensure that messages which do not require an explicit response from the peer are completed on the tx event added by M|OFED to the completion queue. And it also serves to ensure that any messages which require an explicit reply to be completed receive that reply within the tx_timout.

PUT and GET in Routed Configuration

When a node receives a PUT request, the O2IBLND calls lnet_parse() to deal with it. lnet_parse() calls lnet_parse_put(), which matches the MD and initiates a receive. This ends up calling into the LND, kiblnd_recv(), which would send an IBLND_MSG_PUT_[ACK|NAK]. This allows the sending peer LND to know that the PUT has been received, and let go of it's TX, as shown below. On receipt of the ACK|NAK, the peer sends a IBLND_MSG_PUT_DONE, and initates the RDMA operation. Once the tx completes, kiblnd_tx_done() is called which will then call lnet_finalize(). For the PUT, LNet will end sending an LNET_MSG_ACK, if it needs to (look at lnet_parse_put() for the condition on which LNET_MSG_ACK is sent).

In the case of a GET, on receipt of IBLND_MSG_GET_REQ, lnet_parse() -> lnet_parse_get() -> kiblnd_recv(). If a there is data to be sent back, then the LND sends and RDMA operation with IBLND_MSG_GET_DONE, or just the DONE.

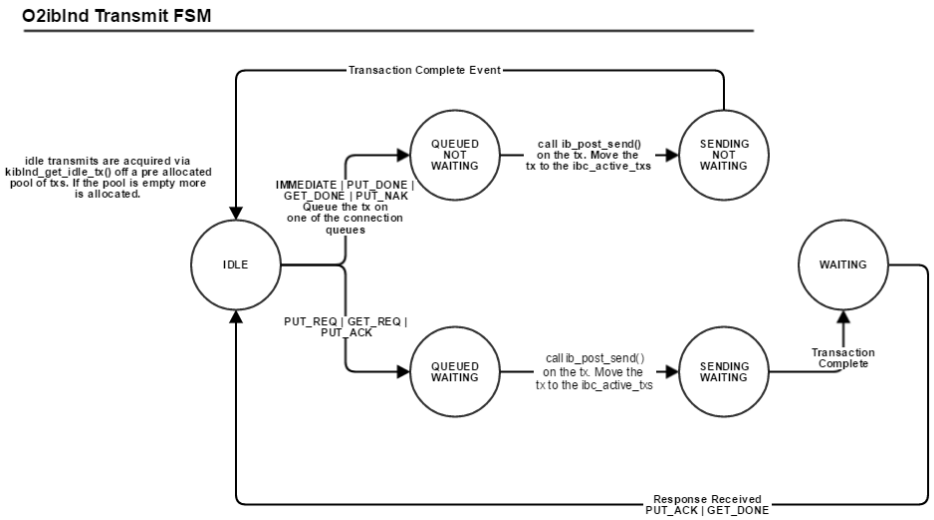
The point I'm trying to illustrate here is that there are two levels of messages. There are the LND messages which confirm that a single LNET message has been received by the peer. And there there are the LNet level messages, such as LNET_MSG_ACK and LNET_MSG_REPLY. These two in particular are in response to the LNET_MSG_PUT and LNET_MSG_GET respectively. At the LND level IBLND_MSG_IMMEDIATE is used.

In a routed configuration, the entire LND handshake between the peer and the router is completed. However the LNET level messages like LNET_MSG_ACK and LNET_MSG_REPLY are sent by the final destination, and not by the router. The router simply forwards on the message it receives.

The question that the design needs to answer is this: Should LNet be concerned with resending messages if LNET_MSG_ACK or LNET_MSG_REPLY are not received for LNET_MSG_PUT and LNET_MSG_GET respectively?

At this point (pending further discussion) it is my opinion that it should not. I argue that the decision to get LNET to send the LNET_MSG_ACK or LNET_MSG_REPLY implicitly is actually a poor one. These messages are in direct respons to direct requests by upper layers like RPC. What should've been happening is that when LNET receives an LNET_MSG_[PUT|GET], an event should be generated to the requesting layer, and the requesting layer should be doing another call to LNet, to send the LNET_MSG_[ACK|REPLY]. Maybe it was done that way in order no to hold on resources more than it should, but symantically these messages should belong to the upper layer. Furthermore, the events generated by these messages are used by the upper layer to determine when to do the resends of the PUT/GET. For these reasons I believe that it is a sound decision to only task LNet with attempting to send an LNet message over a different local_ni/peer_ni only if this message is not received by the remote end. This situation is caught by the tx_timeout.

O2IBLND TX Lifecycle



In order to understand fully how the LND transmit timeout can be used for resends, we need to have an understanding of the transmit life cycle shown above.

This shows that the timeout depends on the type of request being sent. If the request expects a response back then the tx_timeout covers the entire transaction lifetime. Otherwise it covers up until the transmit complete event is queued on the completion queue.

Currently, if the transmit timeout is triggered the connection is closed to ensure that all RDMA operations have ceased. LNet is notified on error and if the modprobe parameter auto_down is set (which it is by default) the peer is marked down. In lnet_select_pathway() lnet_post_send_locked() is called. One of the checks it does is to make sure that the peer we're trying to send to is alive. If not, message is dropped and -EHOSTUNREACH is returned up the call chain.

In lnet_select_pathway() if lnet_post_send_locked() fails, then we ought to marke the health of the peer and attempt to select a different peer_ni to send to.

NOTE, currently we don't know why the peer_ni is marked down. As mentioned above the tx_timeout could be triggered for several reasons. Some reasons indicate a problem on the peer side, IE not receiving a response or a transmit complete. Other reasons could indicate local problems, for example the tx never leaves the queued state. Depending on the reason for the tx_timeout LNet should react differently in it's next round of interface selection.

Peer timeout and recovery model

- On transmit timeout kiblnd notifies LNet that the peer has closed due to an error. This goes through the lnet_notify path.
- The peer aliveness at the LNet layer is set to 0 (dead), and the last alive
- In IBLND whenever a message is received successfully, transmitted successfully or a connection is completed (whether it is successful or has been rejected) then the last alive time of the peer is set.
  At the LNet layer for a non router node, lnet_peer_aliveness_enabled() will always return 0:
  - ```
    #define lnet_peer_aliveness_enabled(lp) (the_lnet.ln_routing != 0 && \
    ((lp)->lpni_net) && \
            (lp)->lpni_net->net_tunables.lct_peer_time_out > 0)
    ```
  - In effect, the aliveness of the peer is not considered at all if the node is not a router.
  - This can remain the same since the health of the peer will be considered in lnet_select_pathway() before this is considered.
  - In fact if the logic for the health of the peer is done in lnet_select_pathway(), then the logic in lnet_post_send_locked() can be removed. A peer will always be as healthy as possible by the time the flow hits lnet_post_send_locked()

  If the node is not a router, then a peer will always be tried irregardless of its health. If it is a router then once every second the peer will be queried to see if it's alive or not.
  - TBD: In o2iblnd kiblnd_query looks up the peer and then returns the last_alive of hte peer. However, there is code "if (peer_ni == NULL) kiblnd_launch_tx(ni, NULL, nid)". This code will attempt creating and connecting to the peer, which should allow us to discover if the peer is alive. However, as far as I know peer_ni is never removed from the hash. So if it's already an existing peer which died, then the call to kiblnd_launch_tx() will never be made, and we'll never discover if the peer came back to life.
  - In socklnd, socknal_query() works differently. It actually attempts to connect to the peer again, within a timeout. This leads the router to discover that the peer is healthy and start using it again.
    ```

Health Revisited

There are different scenarios to consider with Health:

1. Asynchronous events which indicate that the card is down
   Immediate failures when sending
   a. Failures reported by the LND
   b. Failures that occur because peer is down. Although this class of failures could be moved into the selection algorithm. IE do not pick peers_nis which are not alive.
   TX timeout cases.
   a. Currently connection is closed and peer is marked down.
   b. This behavior should be enhanced to attempt to resend on a different local NI/peer NI, and mark the health of the NI

TX Timeouts in the presence of LNet Routers

Communication with a router adheres to the above details. Once the current hop is sure that the message has made it to the next hop, LNet shouldn't worry about resends. Resends are only to ensure that the message LNet is tasked to send makes it to the next hop. The upper layer RPC protocol makes sure that RPC messages are retried if necessary.

Each hop's LNet will do a best effort in getting the message to the following hop. Unfortunately, there is no feedback mechanism from a router to the originator to inform the originator that a message has failed to send, but I believe this is unnecessary and will probably increase the complexity of the code and the system in general. Rule of thumb should be that each hop only worries about the immediate next hop.

SOCKLND Detailed Discussion

TBD

Old

Implementation Specifics

Events that are triggered asynchronously, without initiating a message, such as port down, port up, rdma device removed, shall be handled via a new LNet /LND API that shall be added.

In the other cases, `lnet_ni_send()` calls into the LND via the `lnd_send()` callback provided. If the return code is failure `lnet_finalize()` is called to finalize the message.

`lnet_finalize()` takes the return code as an input parameter. The above behavior should be implemented in `lnet_finalize()` since this is the main entry into the LNet module via the LNDs as well.

`lnet_finalize()` detaches the MD in preparation of completing the message. Once the MD is detached it can be re-used. Therefore, if we are to re-send the message then the MD shouldn't be detached at this point.

`lnet_complete_msg_locked()` should be modified to manage the local interface health, and decide whether the message should be resent or not. If the message can not be resent due to no available local interfaces then the MD can be detached and the message can be freed.

Currently `lnet_select_pathway()` iterates through all the local interfaces on a particular peer identified by the NID to send to. In this case we would want to restrict the resend to go to the same `peer_ni`, but on a different local interface.

This approach lends itself to breaking out the selection of the local interface from `lnet_select_pathway()`, leading to the following logic:

```
lnet_ni lnet_get_best_ni(local_net, cur_ni, md_cpt)
{
        local_net = get_local_net(peer_net)
        for each ni in local_net {
                health_value = lnet_local_ni_health(ni)
                /* select the best health value */
                if (health_value < best_health_value)
                        continue
                distance = get_distance(md_cpt, dev_cpt)
                /* select the shortest distance to the MD */
                if (distance < lnet_numa_range)
                        distance = lnet_numa_range
                if (distance > shortest_distance)
                        continue
                else if distance < shortest_distance
                        distance = shortest_distance
                /* select based on the most available credits */
                else if ni_credits < best_credits
                        continue
                /* if all is equal select based on round robin */
```

```
                else if ni_credits == best_credits
                        if best_ni->ni_seq <= ni->ni_seq
                                continue
        }
}

/*
 * lnet_select_pathway() will be modified to add a peer_nid parameter.
 * This parameter indicates that the peer_ni is predetermined, and is
 * identified by the NID provided. The peer_nid parameter is the
 * next-hop NID, which can be the final destination or the next-hop
 * router. If that peer_NID is not healthy then another peer_NID is
 * selected as per the current algorithm. This will force the
 * algorithm to prefer the peer_ni which was selected in the initial
 * message sending. The peer_ni NID is stored in the message. This
 * new parameter extends the concept of the src_nid, which is provided
 * to lnet_select_pathway() to inform it that the local NI is
 * predetermined.
 */

/* on resend */
enum lnet_error_type {
        LNET_LOCAL_NI_DOWN, /* don't use this NI until you get an UP */
        LNET_LOCAL_NI_UP, /* start using this NI */
        LNET_LOCAL_NI_SEND_TIMEOUT, /* demerit this NI so it's not selected immediately, provided there are
other healthy interfaces */
        LNET_PEER_NI_NO_LISTENER, /* there is no remote listener. demerit the peer_ni and try another NI */
        LNET_PEER_NI_ADDR_ERROR, /* The address for the peer_ni is wrong. Don't use this peer_NI */
        LNET_PEER_NI_UNREACHABLE, /* temporarily don't use the peer NI */
        LNET_PEER_NI_CONNECT_ERROR, /* temporarily don't use the peer NI */
        LNET_PEER_NI_CONNECTION_REJECTED /* temporarily don't use the peer NI */
};

static int lnet_handle_send_failure_locked(msg, local_nid, status)
{
        switch (status)
        /*
     * LNET_LOCAL_NI_DOWN can be received without a message being sent.
     * In this case msg == NULL and it is sufficient to update the health
     * of the local NI
     */
        case LNET_LOCAL_NI_DOWN:
                 LASSERT(!msg);
                local_ni = lnet_get_local_ni(msg->local_nid)
                if (!local_ni)
                                return
                /* flag local NI down */
                lnet_set_local_ni_health(DOWN)
                break;
        case LNET_LOCAL_NI_UP:
                LASSERT(!msg);
                local_ni = lnet_get_local_ni(msg->local_nid)
                if (!local_ni)
                                return
                /* flag local NI down */
                lnet_set_local_ni_health(UP)
                /* This NI will be a candidate for selection in the next message send */
        break;
...
}

static int lnet_complete_msg_locked(msg, cpt)
{
        status = msg->msg_ev.status
        if (status != 0)
                rc = lnet_handle_send_failure_locked(msg, status)
                if rc == 0
                                return
        /* continue as currently done */
}
```

Remote Interface Failure

A remote interface can be considered problematic under multiple scenarios:

- Address is wrong
- Route can not be determined
- Connection can not be established
- Connection was rejected due to incompatible parameters

Desired Behavior

When a remote interface fails the following actions take place:

1. The remote interface health is updated
2. Failure statistics incremented
3. A resend is issued on a different remote interface if there is one available.
4. If no other remote interface is present then the send fails.

There are several ways a remote interface can recover:

1. The remote interface is retried as a destination because there is no alternative available, and no error results.
2. The remote interface is periodically probed by a helper thread. An interesting wrinkle is that there is no reason to probe the remote interface unless there is traffic flowing to the peer through other paths. (LNet doesn't care about the state of a remote node that the local node isn't talking to anyway.)

Implementation Specifics

In all these cases a different `peer_ni` should be tried if one exists. `lnet_select_pathway()` already takes `src_nid` as a parameter. When resending due to one of these failures `src_nid` will be set to the `src_nid` in the message that is being resent.

```
static int lnet_handle_send_failure_locked(msg, local_nid, status)
{
        switch (status)
...
        case LNET_PEER_NI_ADDR_ERROR:
                lpni->stats.stat_addr_err++
                goto peer_ni_resend
        case LNET_PEER_NI_UNREACHABLE:
                lpni->stats.stat_unreacheable++
                goto peer_ni_resend
        case LNET_PEER_NI_CONNECT_ERROR:
                lpni->stats.stat_connect_err++
                goto peer_ni_resend
        case LNET_PEER_NI_CONNECTION_REJECTED:
                lpni->stats.stat_connect_rej++
                goto peer_ni_resend
        default:
                /* unexpected failure. failing message */
                return

peer_ni_resend
        lnet_send(msg, src_nid)
}
```