# 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 37.5 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 */
};
```

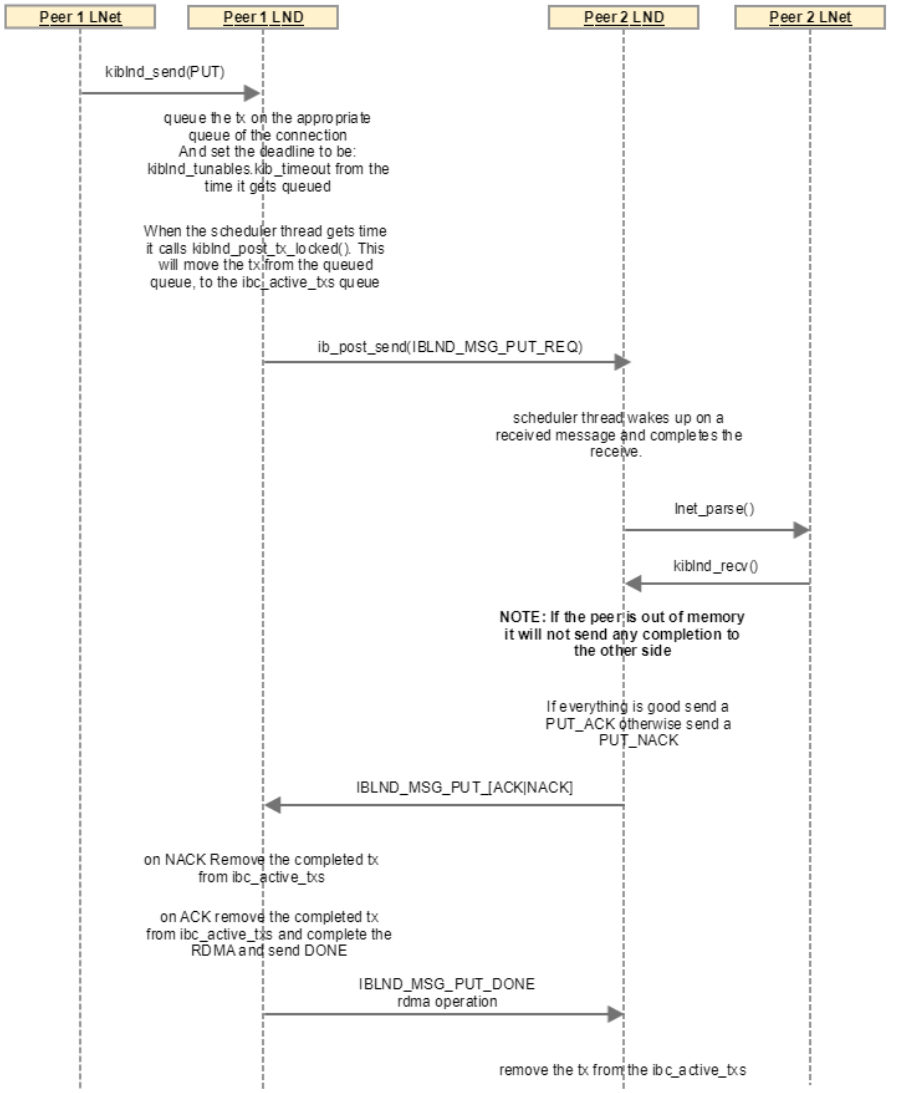
- Although some of the actions LNet will take is the same for different errors, it's still a good idea to keep them separate for statistics and logging.
- on LNET_LOCAL_NI_DOWN set the ni_state to STATE_FAILED. In the selection algorithm this NI will not be picked.
- on LNET_LOCAL_NI_UP set the ni_state 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.>

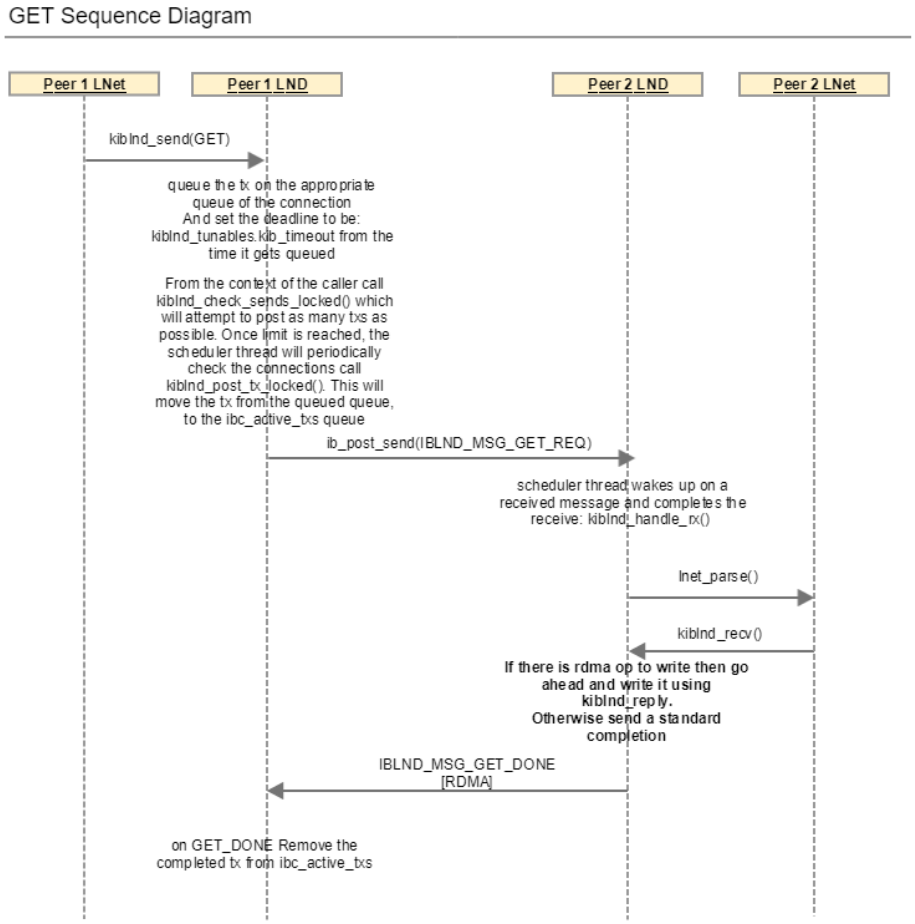
TX timeout is exactly what we need to determine if the message has been transmitted successfully to the remote side. If it has not been transmitted successfully we can attempt to resend it on different peer_nis until we're either successful or we've exhausted all of the peer_nis.

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.

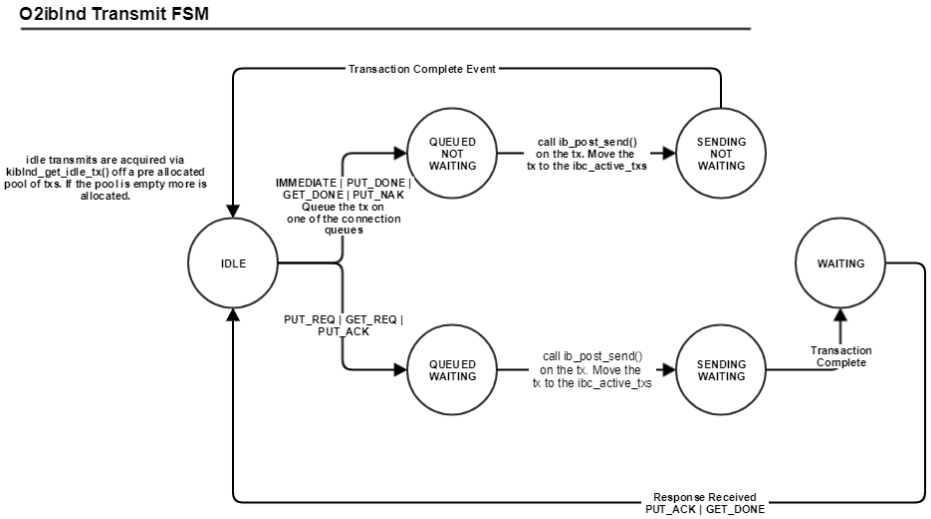
The point I'm trying to illustrate here is that there are two levels of messages. There are the LND messages which confirm that a single LNET message has been received by the peer. And there there are the LNet level messages, such as LNET_MSG_ACK and LNET_MSG_REPLY. These two in particular are in response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