LNet Channel Bonding Design

Author	Date	Description of Document Change	Client Approval By	Client Approval Date
Isaac Huang	05/06/09	Initial draft		
	06/03/09	Edits by John Dawson		
	06/15/09	JKD- Copyright notice		

1. Requirements

To aggregate bandwidth and/or improve overall network availability, LNet should be able to make use of multiple NIs, in a generic way, independent of LND types, as a single logical NI in various modes, which should include standby and load balancing (with various balancing policies like round robin and dynamic congestion avoidance).

The slave NIs of a bonding NI could be of different types (e.g. a TCP NI and an IB NI), or could be of a same type but belongs to a same or different networks (e.g. 10.0.0.1@o2ib0 and 192.168.0.1@o2ib1, or 10.0.0.1@o2ib0 and 10.0.0.2@o2ib0). Note that currently only the TCP LND and the OFA IB LND supports multiple instances, and Portals LND support is already on the way. However, there are two restrictions on slave NIs:

1. Bonding is not recursive, so a bonding NI can't act as a slave NI.
2. Obviously the loopback NI (@lo) can not act as a slave NI.

A bonding NI should know the status of all its slave NIs and stop using a failed NI, and the status of this slave NI should be available to peers so that they could avoid the failed remote slave NIs as well.

Each slave NI should still work independently while serving as a slave, for example, 10.0.0.1@o2ib0 should work properly on itself while acting as a slave NI of a bond NI. This would be important for support rolling upgrades.

A bonding NI may have only one slave NI. For instance, if clients have one NI in @o2ib0 and routers have two NIs in @o2ib0, allowing clients to create a bonding NI over its only o2ib0 slave NI would enable them to reside in the same bonding network as the bonding NIs of routers and balance load among the two o2iblnd NIs of each router.

In a bonding network, there is no requirement that all bonding NIs have identical sets of slave NIs, e.g. some may have two IB slave NIs and some may only have one. However, it is required that all bonding NIs should share at least one slave NI network – i.e. all of them should have direct connectivity, which is a fundamental attribute of a LNet network.

Users may opt to bond different kinds of NIs together, e.g. a o2iblnd NI and a

socklnd NI, in whatever mode they like. Users shall be aware of the consequences (e.g. potentially high CPU usage when balancing load over IB and TCP links).

The channel bonding mechanism should be transparent to upper layers (i.e. PTLRPC and Lustre services) – no code change would be needed for upper layers.

Finally, channel bonding is not supported in user space lnet.

2. Design Overview

2.1 Terminology

Bonding NI: an instance of the bond LND, which abstracts several physical NIs into a single logical NI. Also known as master NI or parent NI.

Bonding network: an LNet network consisted of bonding NIs.

Slave NI: a physical NI that works under a bond NI.

Bonding Data Server: abbrev. BDS, a node in a bonding network that resolves bonding NIDs into NIDs of their slave NIs. It also provides information like the bonding mode of a bonding NID.

2.2 Overview

2.2.1 The Need For A New LND Driver

An NID is required for a set of slave NIs bound together, for transparency to the upper layer protocols. Consider the two cases below, without a bonding NID:

1. For each outgoing RPC to a server, a client posts its reply and bulk buffers in such a way that only the server could access them. But now there's no way to know in advance which slave NI and thus source NID the reply and the bulk messages of the server would come from. It's possible to alter the LNet API so that the upper layer could grant access to a set of NIDs instead of only a single NID, but it defeats our goal of transparency in two ways: first, it needs an API change and thus upper layer code changes; second, the knowledge of slave NI membership has to be propagated to the upper layers. Other alternatives either break transparency or come with a high performance impact.

2. There's no way for the upper layer to precisely specify which interface to use for an outgoing message. LNet determines outgoing interface based on destination NID and source NID of a message. Since there's no NID for either the source bonding NI or the destination bonding NI, it's impossible for LNet to figure out whether an outgoing message should be handled by the bonding driver or slave NI driver. For backward compatibility and support of rolling upgrades, it's important for a server to be able to communicate with old clients via a slave NI directly or via a bonding NI with new clients.

The requirement for a new kind of NIDs necessitates the need for a new LND driver. Moreover, it's a natural implementation choice to develop LNet channel bonding as a new LND driver, since channel bonding creates a new kind of interface. Other benefits of a new LND driver include total freedom to define the LND-level protocol, independent from the protocols of slave NIs and the LNet layer, and simplification of routing configurations.

However, like the loopback LND driver, the bonding driver is special and tightly coupled with the LNet layer and shall be built as part of the lnet kernel module - LNet needs to directly call bonding driver routines that are not part of the LND API; moreover, because the bonding LND runs over different kinds of physical wire protocols of slave NIs, it has to make use of LNet external APIs to implement its own protocol. For example, it may use LNetPut to exchange protocol data with peers or bonding data servers. The implementation is not strictly at the LND level and the code shall reside in the core lnet directory.

2.2.2 Bonding Modes

Two bonding modes are being considered.

Standby mode: only one slave NI is active at a time. Communications should failover to a backup NI when the default NI fails, and failover to the next backup NI if the current active NI fails again. When the default NI comes back to life, switch back to default NI again or stay with the current active NI but would prefer the default NI when a next failover is necessary (depending on standby mode options). The switch-back behavior is useful when a slow link is used as backup only when the default fast link isn't working.

Load balancing mode: communication is distributed over all slave NIs. The traffic distribution pattern is determined by load balancing policy. LNet doesn't guarantee ordered delivery, so we don't have to worry about messages being reordered by different paths. Load balancing policies should include round-robin, static weight round robin, and dynamic weight round robin (weight adjusted at run

time, probably based on perceived bandwidth).

2.2.3 Configuration Syntax

The bonding NI is configured as other conventional NIs, i.e. via lnet module option 'ip2nets' or 'networks'. For example:

options lnet networks="tcp0(eth0), tcp1(eth1), bond0(*@tcp0, *@tcp1, mode=loadbalance)"

Or:

options lnet ip2nets="tcp0(eth0) 192.168.0.*; tcp1(eth1) 10.1.1.*; bond0(@tcp0, @tcp1) 192.168.0.*"

The 'mode' option shown above is a per-NI option that only applies to NI 'bond0'. In fact, most LND options should really apply to a single NI (e.g. credits, timeout):

options networks="tcp0(eth0, credits=256, peer_credits=16), tcp1(eth1, credits=128)"

Global parameters of the bonding LND driver itself are specified via the driver module's options, e.g. options kbondlnd foo=1. Such parameters affects all bonding NIs.

LNet currently does not support per-NI options, so a new syntax needs to be added: options given inside the brackets of a NI's interface specification should apply to that NI only. Options of the LND driver are global and apply to all instances of the LND. Without per-NI option support, it would be impossible to create multiple bonding NIs that run with different options like bonding modes.

Ip2nets-like wildcard matching is necessary to create cluster wide configurations, which makes administration much easier. For example:

options lnet networks="tcp0(eth0), tcp1(eth1), bond0(*@tcp0, *@tcp1)"

The matching facility is also useful where slave NI NIDs might not be completely static (e.g. in a TCP network where client IPs are assigned dynamically by DHCP).

The NID of a bonding NI is: ADDR + @bond[0-9], where ADDR is LNET_NIDADDR(NID of 1st slave NI). For instance, 192.168.0.11@bond0 is the NID of a bonding NI whose first slave NI is 192.168.0.11@tcp0, or 11@bond0 for a bond NI whose first slave NI is 11@ptl0.

Since the ADDR part of a bonding NI NID depends dynamically on slave NI NID format, special handling must be added for bonding NIDs in 'libcfs_nid2str' for proper string conversion.

The first slave NIs of all bonding NIs in a bond network must belong to a same slave network – otherwise the NID of a bonding NI could not be guaranteed to be unique within a bonding network; moreover, a degraded bonding network (when bonding data servers are not working properly) would make use first slave NIs to resolve bonding NIDs. For example, it's incorrect to have 'bond0(*@o2ib0, *@o2ib1)' on one node, and 'bond0(*@o2ib1, *@o2ib0)' on another. However, LNet can't detect such errors; it's the user's responsibility to set up bonding options properly.

2.2.3.1 Dynamic Interface Configuration

There is a serious limit in the LNet configuration mechanism of having everything inside a string module option – the Linux kernel has a hard limit on the length of string module options, which is by default 1024 bytes. Furthermore, as configurations become more complicated, users would like to add more comments, which makes it easier to hit the kernel hard limit even on a small cluster.

Dynamic interface configuration based on user space utilities (e.g. like what 'ifconfig' does to inet interfaces) would be very desirable.

2.2.4 Bonding Data Server

A BDS resolves bonding NIDs into slave NIDs and the bonding mode, e.g. 192.168.0.11@bond0 into (192.168.0.11@tcp0, 10.1.1.11@tcp1, mode=loadbalance). All bonding NIs register their bonding configuration at the bonding data servers of the bonding network at initialization time. Bonding data servers are specified via per-NI options:

options lnet networks="bond0(@o2ib0, @o2ib1, bds=192.168.1.1@o2ib0,10.1.1.1@o2ib1)"

Before any communication to a bonding NID can happen, it must be resolved into

its slave NIDs.

Number of slave NIs under a bonding NI is theoretically only limited by the LNET_MTU, the maximum number of bytes a bonding protocol message could carry. Of course in practice you can't put that many interfaces in a single host.

Note that BDS only resolves bonding NIDs and does not provide health status of slave NIs, therefore the load on a BDS should be rather low.

3. Monitor Slave NI Status

In Linux ethernet bonding, the bonding driver monitors local slave interface status by querying link-level interface status or pinging reference nodes. In case of status changes, update switch states (e.g. forwarding table) via LACP or gratuitous ARP.

In LNet channel bonding, slave LNDs are at a rather high level in the protocol stack, so it is hard to figure out the status of a low-level interface (e.g. ethernet NIC or IB HCA) by querying its link-level state. We could resort to the local slave NI's last_send and last_recv timestamps or pinging reference nodes (e.g. the bonding data servers).

Assuming that local slave NI status could be reliably detected, there's the much more difficult problem of propagating local slave status changes to peers in a scalable manner. In ethernet bonding, the switches are the natural places for such data since all packets flow through switches, and it could be transferred efficiently by multicast. Unfortunately, there is no such place in LNet. Local NI status changes have to go through bonding data servers and broadcasted or multicasted across the network.

Instead of monitoring local slave NI status, it's also possible to monitor remote slave NI status of bonding peers. In fact, work in the shared routing project has already laid down infrastructure for NIs to detect peer health and report it to LNet. I'm leaning towards making use of peer health detection, instead of monitoring local NI status. Advantages are:

1. Apparently no broadcasting or multicasting is needed, and thus load on the BDS is rather low.

2. Naturally I only monitor peers I intend to communicate with.

3. It's much simpler and reuses existing code.

The bonding driver should adopt a mixed approach that includes both local NI status monitoring and remote NI status detection:

1. It could infer local slave NI status by watching the timestamps of last successful send or receive operation, and stop using it when it appears dead. However, this local health information would not be propagated to the driver's node's peers, because it's very expensive and peers have a better way of detecting the health of the interface and the path to their peers. Peers would send keepalive messages or ping messages to my interfaces to find out their status, so the timestamp of a local interface which I've stopped using would still grow, which serves as an indication of its reviving, despite the fact that I've stopped using it for outgoing traffic.

2. It relies on LND peer health detection to find out remote slave NI health, or more precisely, the health of the path to a peer's slave NI. The LND peer health detection relies on send/recv timestamps and out-of-band keep-alive messages, so it detects both end-point failures and path problems. The most important benefit of using LND peer health detection is its avoidance of a mechanism to propagate timely local NI status to the bonding peers.

3.1 Response Time

It looks attractive to be able to detect remote slave status changes as soon as possible, but:

1. It would increase overhead, and add much more keep alive traffic.

2. It increases false negatives, in the event of congested network.

Moreover, it makes no sense to only find out peer slave NI failure after the local Lustre client has already started using a failover server, and it's very bad not to be able to detect remote NI status before Lustre starts to retransmit a message. So we must find a sweet spot where it plays nicely with Lustre services in a manner that is not too aggressive.

3.2 Multiple Slave In a Same Network

One tricky thing with having multiple slaves in the same network is that a peer

slave NI might appear dead to one slave but not to another, if it's not a remote end-point failure. For example, peer slave 192.168.1.21@o2ib0 may appear dead to local slave 192.168.1.3@o2ib0, but is indeed alive via another local slave NI 192.168.1.4@o2ib0, because there's either a cable failure with my slave 192.168.1.3@o2ib0 or a failure in the path:

Therefore:

1. The local bonding driver must query peer slave status via all the local slave NIs that could reach the peer slave.

2. Local LNDs must keep per-NI peer state, not per LND. For instance, in the example above, local slave NIs must keep two separate peer health states for the same physical peer.

4. Multiple Instances Per LND

Now, three LNDs support multiple instances (i.e. multiple NIs of its LND type): socklnd, o2iblnd, and ptllnd. But it's not enough to support channel bonding. Currently these LNDs keep global peer states, which could be problematic when there are more than one NI in a same network, e.g. two NIs in the @o2ib0 network because of the following:

1. Only one copy of peer health state. So if there's local NI failure, we would still believe that peers could be reached via that NI, when in fact they are only reachable via the other local NI.

2. Only one connection to each peer. To aggregate bandwidth, there must be multiple connections to a single peer, via multiple local NIs. For example, there must be two connections to 192.168.1.11@o2ib0 via my two NIs o2ib0(ib0) and o2ib0(ib1).

The solution is to create per-NI peer structures.

The o2iblnd logic to validate incoming connection requests should also be updated, because it assumes that all its NIs live are in separate LNet networks. There's no such problem with the socklnd.

5. Bonding Data Server

A bonding network must have at least one bonding data server that resides on all

slave networks, and it's involved in the following:

1. When each bonding NI initializes itself, it registers its bonding configuration with the bonding server.

2. When local bonding configuration has been changed (e.g. bonding mode changed by administrator), notify bonding server with updated configuration data.

3. When a bonding driver needs to talk to a bonding peer, it resolves the peer's bonding NID into slave NIDs by querying bonding server.

5.1 Degraded Mode

When no bonding data server is alive, it's still possible to resolve bonding NID's by querying a bonding peer directly; when this happens, we say the network is running in degraded mode.

When a bonding NI wants to query a peer, it knows nothing about the peer's bonding configuration other than its bonding NI NID, so there must be a way to find out the destination NID and source slave NI to use. The trick is in the bonding NI NID. The ADDR part of the NID comes from its first slave NI NID, and it's required that the first slave NIs of all bonding NIs in a same bond network are the same. So destination NID is LNET_NIDADDR(peer_bond_NI) + LNET_NIDNET(my_1st_slave_NID), i.e. NID of peer bonding NI's first slave NI, and should be sent over my first slave NI.

For example, my bonding NI is 'bond0(10.0.0.11@o2ib0, 10.0.0.12@o2ib0)', the initial HELLO to 10.0.0.22@bond0 shall be sent to LNET_NIDADDR(10.0.0.22@bond0) + LNET_NIDNET(10.0.0.11@o2ib0) = 10.0.0.22@o2ib0, via my first slave NI 10.0.0.11@o2ib0.

But there is an important limitation, which is why we'd bother with creating bonding data servers. When a peer's 1st slave NI has already failed or there's a local slave NI failure which prevents me from reaching the peer's 1st slave NI, it's impossible to query a peer directly since I only know their 1st slave NI NID. This is why we call it degraded mode.

In degraded mode, all peers essentially act as bonding data servers that only serve their own bonding configuration data. Therefore the query and the answer are still carried over the BDS protocol, and no new code needed.

5.1 Initial Handshaking Protocol

Before any query or registration could be sent out to a bonding server, a handshaking process must complete in order to negotiate protocol version (for future protocol changes and backward compatibility).

An initial HELLO message contains the following fields:

1. A static predefined magic number, used to determine byte sex (u32) (0x0eeb1958)
2. Source bonding NI (i.e. mine) incarnation stamp (u32).
3. Target bonding NI (i.e. peer)
4. Protocol version data: detailed later
5. My bonding NI configuration data: slave NIs and their status, bond mode (policy if load balancing, default active slave NI if standby backup)

All message fields go on wire in host native endianness; the target peer flips multi-byte fields properly by finding out source peer byte sex via the magic number.

HELLO messages are send to the LNet reserved Portal. HELLO buffers are posted at startup. There's no need to post more at runtime, because we always handle HELLO eagerly and repost the buffer immediately and the reserved portal is marked as a lazy portal

5.1.1 Initial Handshake

The initial handshaking process is started when: LNet asks a bonding NI to send a message to an unknown peer, or bonding NI receives an incoming HELLO from an unknown peer.

Version data starts right after the u32 incarnation stamp. Protocol version data includes a version number (u32) and capability masks (u64 for each). The version number equals the number of capability masks. Each bit in a mask stands for a minor feature supported by the version, so a mask could serve as a minor version number. When all the bits of the current version capability mask

are used up, a new version is created by increasing version number by one. However, the interpretion of the u64 mask is solely determined by the version. For instance, a version may opt to interpret its capability mask as two u32 integers, in order to negotiate integer data (e.g. number of credits).

Total length of version data is therefore: one u32 + 'version' u64s

Normal data can flow only after two nodes have exchanged HELLO messages, at which point the version of the connection is set to the lower version of the two ends (so that peers of newer protocols can speak with old peers).

At the time a new peer structure is created, bonding NI sends a HELLO message over. When peer gets my HELLO, he creates a new peer structure for me and responds with his HELLO. My new peer goes into connected state when his HELLO is received; my peer's peer structure goes into connected mode immediately after he has replied me with a HELLO.

Source incarnation stamp in the initial HELLO is set to the current bonding NI incarnation stamp.

Target incarnation stamp is set to zero if I'm initiating the handshaking process and don't yet know the peer's incarnation, or set to peer's last known incarnation if I'm replying a HELLO message.

5.1.2 Status Updates

Whenever my local bonding configuration changes, e.g. a slave NI has gone down, the bonding NI needs to notify its peers about this change.

The bonding LND maintains a local incarnation stamp for each bonding NI. The incarnation is initialized to a random number in lnd_startup, and increased by 1 every time there's a state change in the NI's bonding configuration.

When a new peer structure is created, record my current NI incarnation stamp in it. Whenever there's a new message from a peer, check its local incarnation stamp, and send it a HELLO update (with my latest bonding configuration inside) if its local incarnation is outdated (i.e. smaller than the current NI incarnation). Lazy notification works for load balancing mode, where peers could still reach me after a local slave NI failure. In standby mode, we'd consider eagerly notify all the peers who still have outdated bonding data.

My local incarnation stamp is also included in each HELLO message sent (no matter whether it's the initial HELLO or not).

Peer's last known incarnation is included in the HELLO's target stamp field.

Whenever I received a HELLO whose target stamp is not zero and doesn't equal my bonding NI's current incarnation stamp, this peer has stale information (I haven't got around to giving him an update, or I have rebooted recently) so I send him a HELLO too.

5.1.3 Send and Recv HELLO messages

The HELLO message is sent to a peer's LNet reserved portal by LNetPut, using static pre-defined matchbits. Each peer posts some HELLO buffers to it's reserved portal that matches the static matchbits from any node during bonding driver initialization. There must not be any match with any buffers posted to this reserved portal.

Bonding driver is notified on incoming HELLOs by a callback function associated with the HELLO buffers. Note that it can't use any LNet API in the callback, otherwise deadlock would happen.

6. Outgoing Messages

6.1 lnet_send()

All LNet messages are sent by function 'lnet_send'; 'lnet_send' should skip 'lnet_post_send_locked' if source NI is a bonding NI, both in local send and routed send (since, unlike the loopback NI, a bonding peer can serve as router). All tx credit accounting is handled by slave NI's credit system when messages are passed to slave NIs.

However, we can't skip lnet_post_routed_recv_locked.

6.2 lnd_send(ni, priv, lnet_msg)

There are no LND-level credits in bonding NIs; they rely on their slave NIs to implement proper flow control mechanisms.

The 'lnd_send' API of bond LND looks up its peer table and:

1. If a resolved peer is found, send the message by bond_send_msg(peer,

lnet_msg)

2. If an unresolved peer is found, queue the message to the peer.

3. If no peer is found, create one, start the bonding NID resolving process (querying bonding server), and queue the message to the new peer structure.

RDMA and optimized GET are completely determined by slave NIs – the bonding NI just passes the baton to a slave NI (see flowing sections for details).

6.4 bond_send_msg(peer, lnet_msg)

The 'bond_send_msg' function choose a destination slave NID and a local slave NI to send the message, and hand the message over to the slave NI:

1. Choose a destination NI, i.e. peer's slave NI. Look at the peer's bonding configuration: for load balance mode, choose the next (based on load balance policy) slave NI which is up and reachable from me directly (i.e. no router needed in between). If no such target slave NI exists, fail immediately. For standby mode, choose the current active slave NI. Fail immediately if the chosen NI can't be reached directly via any of my own slave NIs. Replace msg_target.nid of the lnet_msg with NID of destination slave NI. Set msg_txpeer of the 'lnet_msg' to peer structure of the destination NI.

2. Choose source NI, i.e. my slave NI. If the bonding NI has only one slave that can reach destination NI directly, choose it. Otherwise, choose the next one for load balancing mode, and favor the active slave NI for standby mode.

3. Hand the message to the chosen slave NI. Call 'lnet_post_send_locked' to send the message. Since the message will be handed over to msg_txpeer.lp_ni of the lnet_msg, 'lnet_post_send_locked' can't be skipped now because slave NI needs to perform tx credit handling.

4. Message completion. Once an lnet_msg has been handed to a slave NI, the bonding NI forgets about it completely and keeps no reference to it. The lnet_msg itself also has no reference to the bonding NI, its msg_tx_peer belonging to the slave NI. When slave NI finalizes the message, all tx credits will be returned properly to slave NI, and all router buffer credits will be returned properly to the peer slave NI (msg_rx_peer always belongs to slave NI network).

7. Incoming Messages

Slave NIs call 'lnet_parse' directly, so master NI won't have a chance to look at the messages. 'lnet_parse' should call a bonding driver function to notify bonding NI about incoming messages.

Slave NIs are responsible for receiving incoming messages, so the lnd_recv function of bonding LND shall never be called.

7.1 HELLO Messages

The bonding driver uses a dedicated EQ when posting HELLO buffers to the reserved portal, LNet invokes the EQ callback on each incoming HELLO message. The callback function schedules the HELLO to be processed, since it can't call lnet APIs from inside EQ callbacks.

Sanity checks are performed first: correct magic, source incarnation stamp not zero, version number not zero. Then flip bytes depending on peer byte sex.

If the HELLO is from an unknown peer, create a new peer structure, save its bonding configuration, reply with a HELLO, and move peer state into connected. However, it may or may not be the initial handshaking HELLO, depending on its target incarnation stamp:

1. Zero: initial handshake HELLO.

2. Nonzero: I may have rebooted recently, peer still has old my state.

In both cases, my reply HELLO will refresh or initialize peer's copy of my bonding state.

If it's from a known peer in unconnected state (i.e. HELLO sent), and its target incarnation stamp:

1. Equals my bonding NI's incarnation: it's a reply for my HELLO, save peer bonding configuration and move peer state into connected.

2. Equals zero: it's a connection race – the peer had sent his HELLO before my HELLO arrived at his door. Save peer bonding configuration and move peer state into connected.

3. Otherwise: I may have rebooted recently, and the peer still has old connection

and sends me a status update. Just save its bonding configuration and move peer state into connected. My HELLO will bring peer's data up to date.

Whenever peer state goes into "connected": set peer's protocol version number to the smaller one of version number in peer's HELLO message and my own version number. If the peer's version is higher, just ignore all capability masks of higher versions since the peer won't make use of any new protocol features when talking with me.

If it's from a known peer in connected state, and its target incarnation stamp:

1. Equals zero: it's an initial HELLO message - peer must have rebooted, save peer bonding data, and reply with a HELLO.

2. Equals my bonding NI's incarnation: it's a status change notification, just save peer bonding data.

3. Otherwise: the peer has stale bonding data of mine, save the peer's bond data and send him a HELLO to update his copy of my bonding data.

7.2 Normal Messages

Slave NIs receive new messages and call 'lnet_parse' to process them. The parent NI is not involved with handling incoming messages at all.

LNet may or may not know whether a message is destined to a slave NI or its parent bond NI:

1. If I'm the final destination of the message, lnet_parse can figure out whether it's to a slave NI or the parent by simply looking at the message's dest_nid field.

2. Otherwise, I'm routing the message to its final destination, and there's really no way to find out whether the last hop sent this message via a bonding NI or a non-bonding NI.

Whenever LNet knows for sure a message to a bonding NI has arrived, it should notify the bonding NI (bonding NI may need to do some sanity checks, e.g. has a peer sent me a HELLO already?).

The 'from_nid' argument of 'lnet_parse' is always set by a slave NI or a non-bond NI, so it's always a slave NI of my peer if it's over a bond network. As a consequence, msg_rx_peer of incoming messages are always peer's slave NI. So router buffer accounting is done according to slave NI credits and peers, and parent NI is not involved at all.

8. Startup and Shutdown

LNet should initialize all its slave NIs before initializing a bonding NI.

LNet should treat bonding NI like the loopback NI when sending messages and disregard peer TX credits and NI TX credits. Moreover, router buffer credits accounting is done via slave NI peers (4.2 Normal Messages). Therefore, 'lnd_startup' could leave bond NI's ni->ni_peertxcredits and ni->ni_txcredits uninitialized – LNet is never going to make use of them.

Initialize the NI incarnation stamp with a random number.

On shutdown, LNet must tear down bonding NI before closing any of its slave NIs.

8.1 base_startup

Called by lnd_startup for one-time global initializations: create EQ, post HELLO buffers, create peer hash table, mark the lnet reserved portal as lazy, and so on.

8.2 base_shutdown

Called by lnd_shutdown when the last bond NI is being closed. In short, undo base_startup.

9. Backward Compatibility and Rolling Upgrade

Version number and capability masks in handshaking HELLOs could solve the backward compatibility between initial version of the bonding protocol with future versions.

The problem of rolling upgrade a cluster from without bonding driver to bonding enabled configuration could be solved as:

1. Rolling upgrade of everyone's code to include bonding driver.

2. Upgrade everyone's configurations to include bonding NIs.

3. Change server NIDs to bonding NIDs in MGS configuration.

4. Clients restart or remount filesystem.

10. Miscellaneous

Run time state should be made available via /proc entries, or the new parameter tree mechanism.

User space utilities should be able to: turn on/off slave NI, failover to a backup NI, and other functions perhaps.