# Multi-Rail High-Level Design

**Document authors** Amir Shehata Olaf Weber
**Designer** Amir Shehata Olaf Weber
**Developers** Amir Shehata Olaf Weber

## Contents

# Introduction

## Objectives

This High Level Design Document outlines the Multi-Rail design in sufficient detail that it can be used as the basis for implementation.

The intent for the first revision of this document is to target sign-off by all stakeholders. Subsequently as the implementation work is divided into phases, multiple other documents will be created as needed detailing the design further. This document will be updated with reference links to the other detailed design documents.

## Reference Documents

| Document Link |
| --- |
| Multi-Rail Scope and Requirements Document |

## Document Structure

This document is made up of the following sections:

**Design Overview:** Describes data structures and APIs for both User Space and Kernel Space

**User Space:** Describes the details of user space changes

**Kernel Space**: Describes the details of Kernel Space changes including the Dynamic Discovery Behavior

## Acronym Table

| Acronym | Description |
| --- | --- |
| LNet | Lustre Network |
| NI | Network Interface |
| RPC | Remote Procedure Call |
| FS | File System |
| o2ib | Infiniband Network |
| TCP | Ethernet TCP-layer Network |
| NUMA | Non-Uniform Memory Access |
| RR | Round Robin |
| CPT | CPU Partition |
| CB | Channel Bonding |
| NID | Network Identifier |
| downrev | Node with no Multi-Rail |
| uprev | Node with Multi-Rail |

# Design Overview

## System level

The following diagram illustrates the components affected by this work and how they relate to each other.

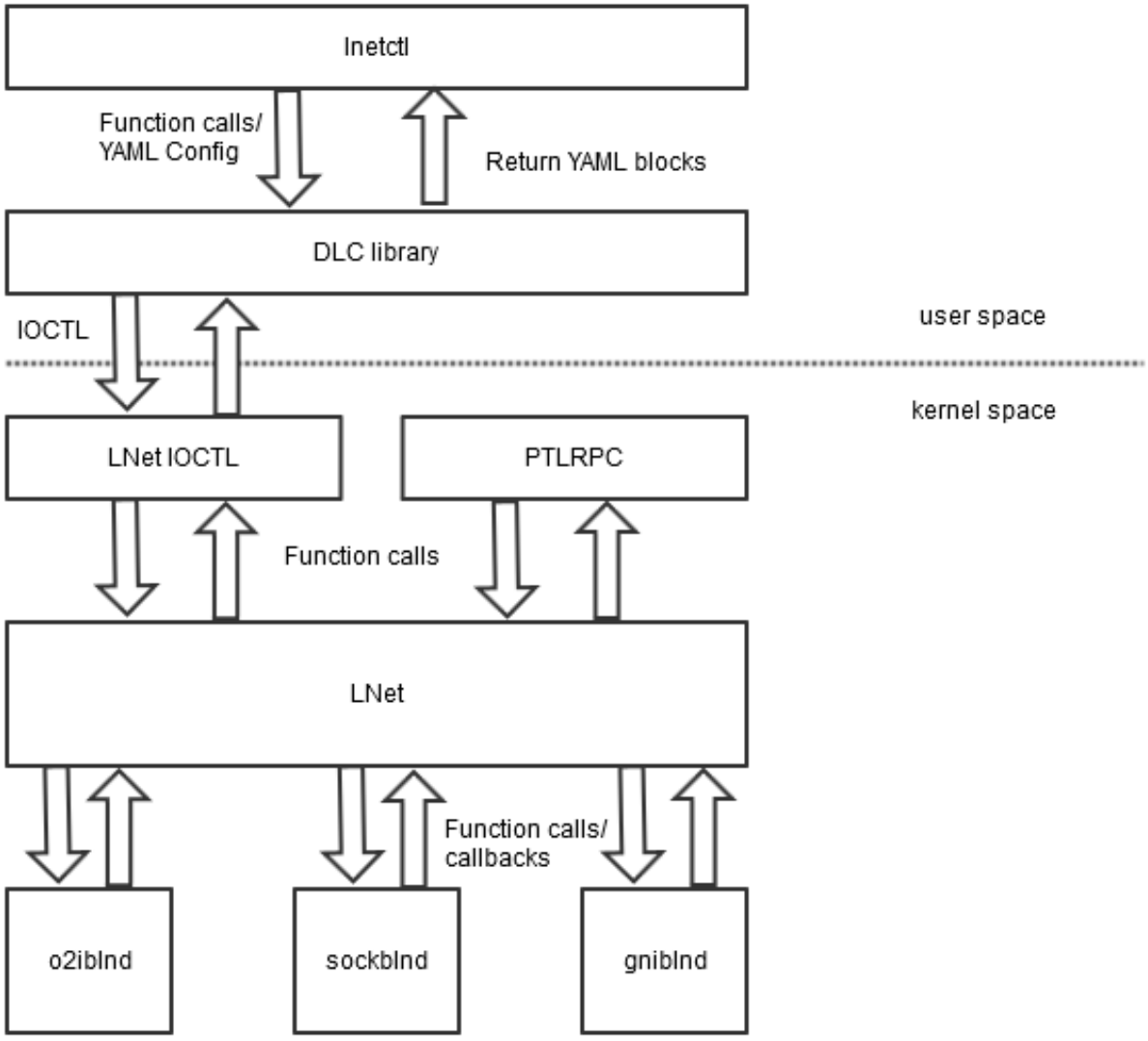



**Figure 1: System Level Diagram**

A quick summary of the changes to be made to the subsystems follows. In addition to the subsystems as such, some changes will be made to the LNet over-the-wire protocol. All of these changes will be discussed in greater detail in following sections of this document.

## `lnetctl`

The `lnetctl` utility will be extended with additional configuration capabilities. Each of the listed capabilities can be configured both via a YAML configuration file and via command-line parameters.

- Define multiple interfaces for the node (local NI or NI). A NI can be both added and removed.
- Define multiple interfaces for a peer (peer NI). A peer NI can be both added to and removed from a peer.
- Define rules that modify how a local NI/peer NI pair is chosen when sending a message. These rules are referred to as User Defined Selection Policies or *selections*.

The changes to `lnetctl` command line and configuration file syntax are discussed below.

# DLC library

The DLC library will be extended to parse the new configuration options, and translate them into the IOCTL calls that communicate with the kernel.

The DLC APIs are described in more details below.

# LNet IOCTL

New IOCTLS are added to handle the new configuration options.

- Add/Delete/Query local NI.
- Add/Delete/Query peer NI.
- Add/Delete/Query selection policies.

A list of the IOCTLS is described in more details below.

# LNet

The primary data structures maintained by the LNet module will be modified as follows: (cfg-040)

- `struct lnet_ni` will reference exactly one NI
- `struct lnet_net` will be added which can point to multiple `lnet_ni` structures
- `struct lnet` (aka `lnet_t`) will have a list of `lnet_net` structures instead of `lnet_ni` structures
- `struct lnet_peer` will be renamed to `struct lnet_peer_ni`, and will represent a peer NID with all its credits
- `struct lnet_peer_net` will encapsulate multiple `lnet_peer_ni` structures. This structure's purpose is to simplify the selection algorithm as discussed later.
- `struct lnet_peer` will encapsulate multiple `lnet_peer_net` structures

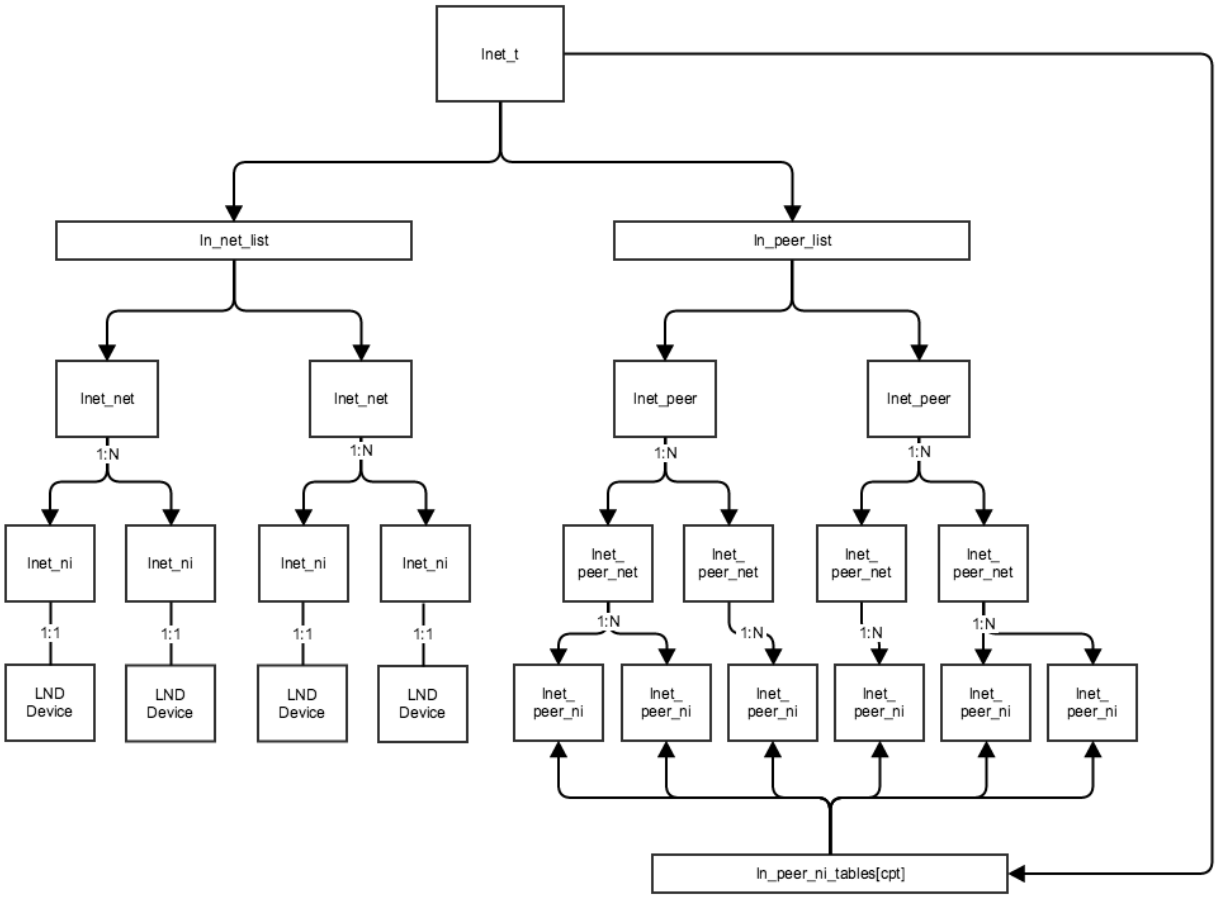



**Figure 2: LNet Data Structure Diagram**

`ln_peer_ni_tables[]` is a hashtable of the `lnet_peer_ni` instances. Since the key to a `peer_ni` is the NID; therefore when messages are received the source NID is used to lookup a `lnet_peer_ni`, and from there a reverse lookup can be done to find the `lnet_peer` structure. Similarly the destination NID can be used for the same purpose when sending a message.

There are operations, such as show commands where the `ln_peer_list` is traversed and the peers are visited and returned to user space to be displayed in YAML format.

Details on how these structures are built are described in the following sections.

# Primary NID

Both LNet and users of LNet like PtlRPC and LDLM assume that a peer is identified by a single NID. In order to minimize the impact of the changes to LNet on its users, a *primary NID* will be selected from a peer's NIDs, and this primary NID will be presented to the users of LNet.

The only hard limitation on the primary NID of a peer is that it must be unique within the cluster. The section on Primary NIDs below goes into more detail.

# PTLRPC

The PtlRPC subsystem will be changed to tell LNet whether the messages it sends to a peer may go over whatever local NI/peer NI combination or whether a specific peer NI should be preferred. The distinction is here between a PtlRPC *request*, which can be sent over whichever path seems most suitable, and a PtlRPC *response* which should be sent to the peer NI from which the request message was received.

The PtlRPC subsystem signals this to LNet by setting the *self* parameter of `LNetGet()`/`LNetPut()` to `LNET_NID_ANY` for a free choice of paths, and to the NID of one of its own interfaces for a restricted choice. The local NID should be the local NID on which the original request was received.

In addition, PtlRPC may be extended to signal LNet to rediscover a peer, for example if it needs to drop the connection to a peer.

# LNDs

No specific changes to the LNDs are planned beyond those necessary to interface correctly with the changes made to LNet.

The multiple-interface support of the `socklnd` layer will be retained for backward compatibility. For testing purposes we may add a tunable to select between interpreting this configuration form as multi-rail and the original behavior. If the tunable is retained in shipping code, it will default to the old behavior. An existing configuration will have the same behavior as before.

# NUMA Selection

An important criterion when selecting a local Network Interface from which to send a message is NUMA locality. When an NI is configured it can be associated with a CPU partition which maps to a NUMA node. The memory used during message sending is allocated on a specific NUMA node. There is significant performance gain in selecting the local NI which is nearest this NUMA node. This can be determined from the CPU partition associated with the local NI on creation.

A "NUMA range" tunable will control the search width. Setting this value to a high number basically turns off NUMA based selection, as all local NIs are considered. cfg-090, snd-025

# Dynamic peer discovery

Dynamic peer discovery will be added to LNet. It works by using an LNet *`PING`* to discover whether a peer node supports the same capabilities. Support is indicated by setting a bit in the `lnet_ping_info_t->pi_features` field.

An LNet *`PUSH`* message is added to enable a node to send its local NI configuration to a peer. To the largest extent possible this will be implemented in terms of the existing interfaces: Event Queues, Memory Descriptors, etc.

Dynamic peer discovery can be enabled, disabled, or verification-only. In the last case, the dynamic discovery protocol will run, but not change peer data structures. Instead it will compare the data

structures and the received information, and complain of differences. Verification can be used to check the validity of YAML configuration files.

In addition it is possible to have situations where dynamic peer discovery is enabled, but some peers have been configured using DLC. We propose to address this case by deferring to the DLC-provided configuration, but also emitting warnings that this configuration differs from what discovery sees. The section on The Discovery Algorithm below goes into more detail.

# Over-the-wire protocol

The LNet `PUSH` message described above is added to the messages that can be sent over the wire. It contains the same data in the same format as an LNet `PING` reply.

It seems likely that the `LNET_PROTO_PING_MATCHBITS` can also be used for the LNet push message – should this turn out to be false, the next available bit will be used.

We add extra information to the `lnet_ping_info_t` datastructure without changing the layout of this structure. The extra information includes the following:

- A feature bit to indicate that the node runs a multi-rail-capable version of the software.
- The NI configuration sequence number.
- If desired it would also be possible to add the LNet version number.

The extra numbers are sent as part of the status information for the loopback NI.

# Use Case scenarios

The description of these scenarios will use *uprev node* as a synonym for a node with a multi-rail capable Lustre version installed. A *downrev node* is a node with an older version of Lustre install, which does not support the multi-rail capability. A *multi-rail node* has the additional interfaces needed to use the multi-rail feature.

Static configurations to be tested include the following, which seem most likely to be encountered in the field:

1. Uprev multi-rail client with downrev servers (MGS/MDS/OSS).
2. Uprev multi-rail servers with downrev clients.
3. Uprev multi-rail clients and servers.
4. Uprev multi-rail clients and servers, with uprev routers.
5. Uprev multi-rail clients and servers, with downrev routers.
6. Uprev multi-rail clients with downrev servers and downrev routers.

Configuration changes that we expect to encounter and which need to be tested:

1. Upgrading a multi-rail client from downrev to uprev, with uprev servers.
2. Downgrading a multi-rail client from uprev to downrev, with uprev servers.
3. Upgrading a router from downrev to uprev
4. Downgrading a router from uprev to downrev

Implicit in the scenarios above is that the full configuration (Net definition, NI definition, Peer NI definition, selection policy definition) is done at once at startup. In addition to this, the following scenarios apply to a cluster that is already up and running:

1. Add a Net, including NIs and Peer NIs.
2. Deleting a Net, NIs and Peer NIs
3. Adding routes
4. Deleting routes
5. Adding selection policies.
6. Deleting selection policies.

# Edge Case scenarios

A mix of edge scenarios that we already can anticipate.

Edge cases for dynamic discovery tend to be race conditions, in particular involving the setup of the datastructures for a peer. Mostly they can be handled by ensuring a `peer_ni` is created and findable early in the process, and marked as initializing. But in some cases we may need to merge partially constructed structures (case 3 below).

1. Two peers attempting to discover each other at the same time.
2. Two processes on a node triggering discovery of a single peer via the same peer NI.
3. Two processes on a node triggering discovery of a single peer via different peer NIs.

Edge cases for `lnetctl` driven configuration tend to involve tearing down in-use datastructures, and inconsistent configuration, especially between nodes.

1. Removing a Peer NI while it is in use. Such an operation is allowed to fail, but we should be able to characterize what "in use" means in that case, and what is required to render the Peer NI idle.
2. Removing a NI while it is in use. Such an operation is allowed to fail, same note applies.
3. Removing a Net while it is in use. Same note applies.
4. Various flavors of having nodes with inconsistent configuration. Maybe we can detect (some) such cases, or at least characterize and document the kind of error messages or bad behavior that will result.

# Debugging Requirements

Since there isn't much to add in terms of High level design regarding the debugging requirements as defined in the scope and requirements document, these requirements will not be detailed further in the design document, but will be implemented in the code.

# User Space

**lnetctl**

The lnetctl utility provides a command line interface. As part of the Multi-Rail project the following commands shall be supported

- Adding/removing/showing Network Interfaces.
- Adding/removing/showing peers. cfg-070, cfg-075
  - Each peer can be composed of one or more peer NIDs
- Adding/removing/showing selection policies

The lnetctl utility uses the DLC library API to perform its functions. Beside the standard command line interface to configure different elements, configuration can be represented in a YAML formatted file. Configuration can also be queried and presented to the user in YAML format. The configuration design philosophy is to ensure that all config which can be queried from the kernel can be fed back into the kernel to get the exact same result. cfg-045, cfg-050, cfg-060, cfg-065, cfg-170

# DLC Library

The DLC library shall add a set of APIs to handle configuring the LNet kernel module. cfg-005, cfg-015

- lustre_lnet_config_ni() - this will be modified to add one or more network interfaces. cfg-020, cfg-025
- lustre_lnet_del_ni() - this will be modified to delete one or more network interface
- lustre_lnet_show_ni() - this will be modified to show all NIs in the network. cfg-010
- lustre_lnet_config_peer() - add a set of peer NIDs
- lustre_lnet_del_peer() - delete a peer NID
- lustre_lnet_show_peers() - shows all peers in the system. Can provide a maximum number of peers to show
- lustre_lnet_config_<rule type>_selection_rule() - add an NI selection policy rule to the existing rules
- lustre_lnet_del_<rule type>_selection_rule() - delete an NI selection policy rule using its assigned ID or matching criteria. cfg-095
- lustre_lnet_<rule type>_selection_rule() - show all NI selection policy rules configured in the system, each given an ID.
- lustre_lnet_set_dynamic_discover() - enable or disable dynamic discovery.
- lustre_lnet_set_use_tcp_bonding() - enable or disable using TCP bonding.

# LNetCtl IOCTL

The following new IOCTLs will be added:

- IOC_LNET_ADD_LOCAL_NI - adds exactly 1 local NI. If the Network doesn't exist then it will implicitly be created. This can be called repeatedly to add more NIs.
- IOC_LNET_DEL_LOCAL_NI - removes exactly 1 local NI. If there are no more NIs in a network the network is removed. This can be called repeatedly to remove more NIs.

- `IOC_LNET_ADD_PEER_NI` - adds a peer NID to an existing peer, if no peer exists with that peer NID a new peer is created
- `IOC_LNET_DEL_PEER_NI` - delete a peer NID from an existing peer, if this is the last peer nid, the peer is deleted.
- `IOC_LNET_ADD_NET_SELECTION_RULE` - add a selection policy rule to identify how to select a network.
- `IOC_LNET_ADD_NID_SELECTION_RULE` - add a selection policy rule to identify how to select a NID
- `IOC_LNET_ADD_CONNECTION_SELECTION_RULE` - add an NI selection policy rule to identify how to select a connection between a local NI and remote NI.
- `IOC_LIBCFS_DEL_<rule type>_SELECTION_RULE` - remove a selection policy rule from the global policy
    - There are two ways to deal with selection policy rules. They can be translated directly into the data structures, but I believe, moreover, they'll need to be maintained separately and applied on new networks which are added later. For example, if you add a o2iblnd network with 4 NIDs. Then you define the priority of this Network via a rule. If you remove and re add this network, you want it to keep the same priority as configured.
    - Also handling the selection policies as a set of rules, from a configuration perspective, is the most intuitive method, since rules can be added, viewed and modified. It makes it easier to view the system configuration as well.
- `IOC_LIBCFS_SHOW_<rule_type>_SELECTION_RULES` - show all the selection policy rules of a specific type.
- `IOC_LIBCFS_SET_DYN_DISCOVERY` - enable/disable dynamic discovery.
- `IOC_LIBCFS_SET_USE_TCP_BONDING` - enable/disable usage of TCP bonding in the system. This affects LNet globally.

## Splitting Adding a Net and Adding an NI

There are two options to consider from configuration perspective

1. Adding a Network separately from adding a Network Interface
    1. This will entail exposing this configuration separation to the user
        1. Advantages
            1. Provides a one-to-one mapping between configuration and internal structure
        2. Disadvantages
            1. Allows the creation of empty networks, which have no use in the system
            2. Adds to the complexity of configuration as the user needs to configure network interfaces in two separate steps
            3. Confuses the functionality. Ex: what happens if the user tries to add an interface without adding a network first? Does the creation fail, or should the network be created anyway?
            4. There is no network configuration specific parameters, except the priority, but even that's configured via the selection policies and not directly.
            5. Configuring a system should have a one-to-one mapping with actual physical changes to the system. The network is a logical construct which is a collection of network interfaces. The network interfaces are what define an I/O point for LNet; therefore a sysadmin should configure a Network Interface and whatever logic that runs in the kernel to get Multi-Rail working should remain hidden from the user.
2. Adding a Network Interface only

1. The user can add a network interface and if the network is not configured it will be created and the network interface is added to it. If the network exists then the Network Interface is added to it.
    1. The advantages and disadvantages are the reverse of the above.

Both the `lnet_net` and `lnet_ni` are not going to be created on a specific CPT, they will simply use `LIBCFS_ALLOC()` to allocate these structures. The CPT association is maintained as a field in these structures, and then used by the LND to allocate its structures. The only two fields which are currently allocated per cpt is `ni->ni_refs` and `ni->ni_tx_queues`. Both of these are allocated on all the CPTs.

Given this, the simplest approach is for user space to send the following information in the same configuration message, or possibly in two separate IOCTLs, but the key point is that the API presented to the user only allows Network Interface configuration:

1. Network to add:
    1. `lustre_lnet_config_net()` takes the network name and interface name.
        1. The DLC library can sanity check that the interface actually exists in the system, before attempting to create the network.
        2. If the interface specified doesn't exist then the kernel would reject this anyway, and no need to send it down.
2. Network Interface to add:
    1. `lustre_lnet_config_net()` takes the network interface name and as mentioned above can do a sanity check to ensure the device is actually configured on the system.
    2. matches the IP address pattern defined in the ip2nets parameters. This can be performed completely in user space since DLC has the same visibility of configured network devices and can perform the matches there. DLC can then proceed to create the exact network interfaces in LNet.
        1. This is an improvement to how ip2net matching currently happens. The current algorithm returns the network and interface irregardless if the IP pattern matches the IP address of the interface identified or not. For example "`tcp1(eth0) 192.168.184.*`" would return `tcp1(eth0)`, even though `eth0` IP is `192.184.182.3`. It would be better if the NI is commissioned only if the IP addresses of the interface matches the IP address pattern
    3. if no explicit interface is configured, but an IP address pattern is present then commission the interface which matches that rule.
        1. Currently the behavior would be to simply return the network name and commission the first interface configured in the system, even though its IP address doesn't match the pattern defined in configuration.

In order to remain backwards compatible, two new IOCTLs will be added to add and remove local NIs. So basically, there will be two ways of adding a network interface.

### Conclusion

Based on feedback at the time of the writing, the approach by which only the configuration of local NI is presented to the user and the addition or removal of a net is implicit, will be preferred.

## Network to Network Interface CPT inheritance

Another open issue is the behavior regarding specifying CPT for Network Interfaces. At the end of the day the CPT is associated with the Network Interface and not with the network. From a configuration perspective there are the following options:

1. CPTs can only be associated with Network Interfaces and not with networks. No configuration option is presented to the user. (recommended approach)
2. A network level CPT list can be specified. That will be resolved at user space in such a way that interfaces with no associated CPT list will use the network level CPT list as the default.
3. The network level CPTs are stored in the kernel and are inherited by Network interfaces added to the network if the network interfaces don't already have an associated CPT list.

The CPTs are creation time element and the best configuration philosophy is to allow the user to explicitly specify it as part of the interface, therefore, it is the recommendation of this design to only allow configuring NI level CPTs. This maintains the current behavior where CPTs are Network Interface specific.

cfg-030 - the CPT is a creation time configuration and can not be changed afterwards. This requirement will not be implemented.

## TCP Bonding vs Multi-Rail

Currently the socklnd implements a form of TCP bonding. The sysadmin can configure a TCP network as follows:

- tcp(eth0,eth1)

This will create a TCP network which bonds both eth0 and eth1 and the socklnd layer.

With the introduction of Multi-Rail, it is desirable to allow the TCP bonding feature to still be usable. However, since Multi-Rail will use the same syntax above to define multiple LNet level network interfaces on the same network, a new configuration value will be introduced to set whether to use TCP bonding or to use Multi-Rail. `use_tcp_bonding` will be a global setting which when set to 1 all TCP networks configured when `use_tcp_bonding` is enabled shall use socklnd bonding over Multi-Rail.
Setting `use_tcp_bonding` will not retroactivly impact already configured TCP networks. Therefore, any attempts to add interfaces to a network which was configured using socklnd bonding, will fail. Otherwise if a network uses Multi-Rail, the addition of network interfaces shall be allowed.

## Userspace configuration Parsing vs in-kernel parser

Multi-Rail will introduce parsing network and peer configuration in the user space DLC Library which will then use IOCTL to create the configuration objects in LNet. The in-kernel parser will remain largely unchanged, except for some slight modifications to allow parsing network interface specific CPTs and other general improvements.

## Backwards Compatibility

Multi-Rail shall change the way network interfaces are configured. In order to maintain backwards compatibility much code will need to be added to deal with different configuration formats. This will inevitably lead to unmaintainable code. As a result multi-rail `lnetctl`/DLC will only work with multi-rail capable LNet. This means that upgrading a system to Multi-Rail capable LNet will entail upgrading all userspace and kernel space components. Older YAML configuration will still work with the newer Multi-Rail capable nodes. bck-005, bck-010, bck-015, bak-20.

Multi-Rail nodes will continue to connect to non-multi-rail capable nodes and vice versa and when a Multi-Rail capable node is connected to a cluster if dynamic discovery is enabled it will automatically be

discovered on first use, as described later in this document in the Dynamic Discovery section. bck-025, bck-030

## Adding local NI

### `lnetctl` Interface

```
# --net no longer needs to be unique, since multiple interfaces can be added to the
same network
# --if: the same interface can be added only once. Moreover it can be defined as a set
of comma
#       separated list of interfaces
#                Ex: eth0, eth1, eth2
lnetctl > net add -h
Usage: net add --net <network> --if <interface> [--peer-timeout <seconds>]
                [--ip2nets <pattern>]
                [--peer-credits <credits>] [--peer-buffer-credits <credits>]
                [--credits <credits>] [--cpt <partition list>]

WHERE

net add: add a network
        --net: net name (e.g. tcp0)
        --if: physical interface (e.g. eth0)
        --ip2net: specify networks based on IP address patterns
        --peer-timeout: time to wait before declaring a peer dead
        --peer-credits: define the max number of inflight messages
        --peer-buffer-credits: the number of buffer credits per peer
        --credits: Network Interface credits
        --cpt: CPU Partitions configured net uses (e.g. [0,1])
```

`--ip2net` parameter can be used to configure multiple Network Interfaces based on an IP address pattern.

Incidentally, the parsing algorithm exists in the kernel, therefore any modifications to the algorithm will benefit both the `lnetctl` utility and the modparams. However no additions to module parameters are being added as part of this project.

Look at the ip2nets section for a more detailed discussion.

### YAML Syntax

cfg-035

```
net:
   - net: <network.  Ex: tcp or o2ib>
     interfaces: <list of interfaces to configure>
                          - intf: <physical interface>
                            CPT: <CPTs associated with the interface>
          detail: <This is only applicable for show command.  1 - output detailed info.
0 - basic output>
     tunables:
        peer_timeout: <Integer. Timeout before consider a peer dead>
        peer_credits: <Integer. Transmit credits for a peer>
        peer_buffer_credits: <Integer. Credits available for receiving messages>
        credits: <Integer.  Network Interface credits>
     seq_no: <integer.  Optional.  User generated, and is
              passed back in the YAML error block>

# Example: configure the o2ib1 network with the ib0 and ib1 interfaces
# this will result in two NIDs being configured on the o2ib1 network:
```

```
#       <ib0-IP>@o2ib1, <ib1-IP>@o2ib1
# ib0 will be associated with CPTs 1 and 3 while ib1 will be associated with CPTs 4
and 5.
net:
   - net: o2ib1
     interfaces:
           - intf: ib0
             CPT: 1,3
           - intf: ib1
             CPT: 4,5

# If no CPTs are configured then by default the interface is associated with all
existing CPTs,
# which is the current behavior.

# It is recommended to use the above syntax rather than the ip2net syntax for clarity.
# The above YAML block will be parsed and the IOCTL structures populated.
```

## DLC API

```
/*
 * lustre_lnet_config_net
 *   Send down an IOCTL to configure a network Interface.
 *
 *   net - the network name
 *   intf - the interface of the network (ex: ib0). This could be a
 *                        comma separated list of interfaces.
 *        - each interface is fed as a separate IOCTL to the kernel.
 *   ip2net - this parameter allows configuring multiple networks.
 *    it takes precedence over the net and intf parameters
 *   peer_to - peer timeout
 *   peer_cr - peer credit
 *   peer_buf_cr - peer buffer credits
 *       - the above are LND tunable parameters and are optional
 *   credits - network interface credits
 *   smp - cpu affinity
 *   seq_no - sequence number of the request
 *   err_rc - [OUT] struct cYAML tree describing the error. Freed by caller
 */
int lustre_lnet_config_net(char *net, char *intf, char *ip2net,
               int peer_to, int peer_cr, int peer_buf_cr,
               int credits, char *smp,
               int seq_no,
               struct cYAML **err_rc);

/* This API will be modified to use IOC_LIBCFS_ADD_LOCAL_NI,
 * instead of the now deprecated IOC_LIBCFS_ADD_NET */
/* Deep error checking is left to the LNet module to perform. An example
 * of deep error checking is checking if an interface that's being added is a
 * duplicate interface */
```

`ip2nets` can be passed as a string to the above API and will be parsed and handled as part of the API.

## DLC API Structures

The following structure is populated and sent down to the kernel. In order to remain backwards compatible with older tools, `lnet_ioctl_config_data` will remain the same, and a new structure will be added for the new way of configuring NIs. The new config NI structure can be extended with LND specific structures which define the tunables. These structures can also be used for sending back information about the NI and the LND tunables back to user space.

```
/*
 * To allow for future enhancements to extend the tunables
 * add a hdr to this structure, so that the version can be set
 * and checked for backwards compatibility. Newer versions of LNet
```

```
 * can still work with older versions of lnetctl. The restriction is
 * that the structure can be added to and not removed from in order
 * not to invalidate older lnetctl utilities. Moreover, the order of
 * fields should remain the same, and new fields appended to the structure
 *
 * That said all existing LND tunables will be added in this structure
 * to avoid future changes.
 */
struct lnet_ioctl_config_o2iblnd_tunables {
        struct libcfs_ioctl_hdr lico_tunable_hdr;
        ...List of all IB tunables...
};

struct lnet_ioctl_config_<lnd>_tunables {
        struct libcfs_ioctl_hdr licn_tunable_hdr;
        ...List of all LND tunables...
};

/*
 * lnet_ioctl_config_ni
 *      This structure describes an NI configuration. There are multiple components
when
 *      configuring an NI: Net, Interfaces, CPT list and LND tunables
 *      A network is passed as a string to the DLC and translated using
libcfs_str2net()
 *      An interface is the name of the system configured interface (ex eth0, ib1)
 *  CPT is the list of CPTS
 *  LND tunables are passed as an extended body
 */
struct lnet_ioctl_config_ni {
        struct libcfs_ioctl_hdr lic_cfg_hdr;
        __u32                                   lic_net;
        char                                    lic_ni_intf[LNET_MAX_STR_LEN];
        __u32                                   lic_cpts[LNET_MAX_SHOW_NUM_CPT];
        char                                    lic_bulk[0]
};
```

## Removing local NI

### lnetctl Interface

```
# In order to remain backward compatible, two forms of the command shall be allowed.
# The first will delete the entire network and all network interfaces under it.
# The second will delete a single network interface

lnetctl > net del -h
net del: delete a network
Usage: net del --net <network> [--if <interface>]

WHERE:

 --net: net name (e.g. tcp0)
 --if: interface name. (e.g. eth0)

# If the --if parameter is specified, then this will specify exactly one NI to delete
or a list
# of NIs, since the --if parameter can be a comma separated list.
# TODO: It is recommended that if the --if is not specified that all the interfaces
are removed.
```

### YAML Syntax

cfg-055

```
net:
   - net: <network.  Ex: tcp or o2ib>
```

```
     interfaces:
         - intf: <interface name to delete>
     seq_no: <integer.  Optional.  User generated, and is
              passed back in the YAML error block>

# Example: delete all network interfaces in o2ib1 network completely
net:
   - net: o2ib1

# delete only one NI
net:
   - net: o2ib1
     interfaces:
           - intf: ib0
           - intf: ib1
```

### DLC API

```
/*
 * lustre_lnet_del_net
 *   Send down an IOCTL to delete a network.
 *
 *   nw - network to delete or delete from
 *   intf - the interfaces to delete. Could be a comma separated list.
 *                        NULL if user wishes to delete the entire network.
 *   seq_no - sequence number of the request
 *   err_rc - [OUT] struct cYAML tree describing the error. Freed by caller
 */
int lustre_lnet_del_net(char *nw, char *intf, int seq_no,
                                  struct cYAML **err_rc);

/* Deep error checking is left to the LNet module to perform. An example
 * of deep error checking is checking if an interface exists before deletion */

/*
 * lustre_lnet_show_net
 *   Send down an IOCTL to show networks.
 *   This function will use the nw paramter to filter the output.  If it's
 *   not provided then all networks are listed.
 *
 *   nw - network to show.  Optional.  Used to filter output. Could be a comma
separated list.
 *   detail - flag to indicate if we require detail output.
 *   seq_no - sequence number of the request
 *   show_rc - [OUT] The show output in YAML.  Must be freed by caller.
 *   err_rc - [OUT] struct cYAML tree describing the error. Freed by caller
 */
int lustre_lnet_show_net(char *nw, int detail, int seq_no,
                                  struct cYAML **show_rc, struct cYAML **err_rc);
```

### DLC API Structures

Same as the above.

## Adding Peer NID

### `lnetctl` Interface

```
lnetctl > peer add -h
Usage: peer add --nid <nid[, nid, ...]>

WHERE:

peer add: add a peer
        --nid: comma separated list of peer nids (e.g. 10.1.1.2@tcp0)
```

The `--nid` parameter can be a comma separated list of NIDs.

The CPT assigned to the peer NID will be specified as part of the `lnet_nid2peer_locked()`.

All peer nids specified must be unique in the system. If a non-unique peer NID is added LNet shall fail the configuration. cfg-080

## YAML Syntax

```
peers:
  - nids:
      0: ip@net1
      1: ip@net2
  - nids:
      0: ip@net3
      1: ip@net4

# The exact same syntax can be used to refresh the peer table. The assumption is
# each peer in the YAML syntax contains all the peer NIDs.
# As an example if a peer is configured as follows:

peers:
  - nids:
     0: 10.2.2.3@ib0
     1: 10.4.4.4@ib1

# Then later you feed the following into the system

peers:
  - nids:
     0: 10.2.2.3@ib0
     1: 10.5.5.5@ib2

# The result of this configuration is the removal of 10.4.4.4@ib1 from
# the peer NID list and the addition of 10.5.5.5@ib2
# In general a peer can be referenced by any of its NIDs. So when configuring all the
NIDs are used
# to find the peer. The first peer that's found will be configured. If the peer NID
being added is
# not unique, then that peer NID is ignored and an error flagged. The Index of the
ignored NID is
# returned to the user space, and is subsequently reported to the user.
```

## DLC API

```
/*
 * lustre_lnet_config_peer_nid
 *   Configure a peer with the peer NIDs
 *
 *   nids - peer NIDs
 *   seq_no - sequence number of the command
 *   err_rc - YAML structure of the resultnatn return code
 */
int lustre_lnet_config_peer_nid(char **nids, int seq_no, struct cYAML **er_rc);
```

## DLC API Structures

```
/* Multiple peers can be defined in the configuration.
 * This will be fed into the kernel as one peer at a time.
 * The first NID in the list will be used as the key NID, and
 * will be passed in every IOCTL to LNet, so that LNet can
 * determine the peer to add the NID to.
 * pr_bulk will be used to pass back peer information to
 * user space.
```

```
 *
 * NOTE: that the first NID to be added is the key NID which means
 * pr_key_nid == pr_cfg_nid;
 */
struct lnet_ioctl_peer_cfg {
    struct libcfs_ioctl_hdr       prcfg_hdr;
        lnet_nid_t                                  prcfg_key_nid;
        lnet_nid_t                                  prcfg_cfg_nid;
        char                                        prcfg_bulk[0];
};
```

LNET_MAX_INTERFACES is the maximum number of NIDs a peer can have. Currently this value is set to 16, and will need to be increased to accomodate the requirement for larger SGI nodes.

## Removing Peer NID

### `lnetctl` Interface

```
lnetctl > peer del -h

WHERE:

peer add: add a peer
        --nid: comma separated list of peer nids (e.g. 10.1.1.2@tcp0)
```

Multiple nids can be deleted by using a comma separated list of NIDs in the `--nid` parameter. All NIDs must be for the same peer.

### YAML Syntax

```
peers:
  - nids:
      0: ip@net1
      1: ip@net2
  - nids:
      0: ip@net3
      1: ip@net4

# This specifies the Peer NIDs that should be deleted. Each grouping of NIDs
# is assumed to be the same NID. The peer is identified by any of its NIDs.
# When a peer is found the NIDs specified for that peer is removed. If the NID
# doesn't exist then an error is outputed and the index of that NID is returned
# to user space, which formats it as a YAML error.
```

### DLC API

```
/*
 * lustre_lnet_del_peer_nid
 *   Delete the peer NIDs. If all peer NIDs of a peer are deleted
 *   then the peer is deleted
 *
 *   nids - peer nids
 *   seq_no - sequence number of the command
 *   err_rc - YAML structure of the resultant return code
 */
int lustre_lnet_del_peer_nid(char **nids, int seq_no, struct cYAML **er_rc);

/*
 * lustre_lnet_show_net
 *   Send down an IOCTL to show peers.
 *   This function will use the nids paramter to filter the output.  If it's
 *   not provided then all peers are listed.
 *
```

```
 *   nids - show only Peer which have these NIDs.
 *   detail - flag to indicate if we require detail output.
 *   seq_no - sequence number of the request
 *   show_rc - [OUT] The show output in YAML.  Must be freed by caller.
 *   err_rc - [OUT] struct cYAML tree describing the error. Freed by caller
 */
int lustre_lnet_show_peer(char **nids, int detail, int seq_no,
                                  struct cYAML **show_rc, struct cYAML **err_rc);
```

### DLC API Structures

```
Same as above
```

## ip2nets

This project will deprecate the kernel parsing of ip2nets. ip2nets patterns will be matched in user space and translated into Network interfaces to be added into the system.

- First interface that matches IP pattern will be used when adding a network interface
- If an interface is explicity specified as well as a pattern, the interface matched using the IP pattern will be sanitized against the explicitly defined interface
    - ex: `tcp(eth0) 192.168.*.3` and there exists in the system `eth0 == 192.158.19.3` and `eth1 == 192.168.3.3`, then configuration will fail, because the pattern contradicts the interface specified.
    - A clear warning will be displayed if inconsistent configuration is encountered.

```
net:
   - net: <net>
         intf: <optional interface>
     pattern: <pattern>

# Example:

net:
  - net: o2ib
    pattern: 192.168.2.*

# If the node has the following IPoIB: 192.168.2.3 and 192.168.2.4, then
# the result of this configuration are the following NIDs:
#   192.168.2.3@o2ib and 192.168.2.4@o2ib
# DLC API will parse the pattern and perform the matching in user space,
# then create the interfaces, after it has sanitized the configuration.
```

## User Defined Selection Policies

One proposal is to define the net and NI priority as part of their creation. However, I'm still leaning toward having the net priority and NI priority as separate rules, stored in a separate data structure. Once they are configured they can be applied to the networks. The advantage of that is that rules are not strictly tied to the internal constructs, but can be applied whenever the internal constructs are created and if the internal constructs are deleted then they remain and can be automatically applied at a future time.

This makes configuration easy since a set of rules can be defined, like "all IB networks priority 1", "all Gemini networks priority 2", etc, and when a network is added, it automatically inherits these rules.

Selection policy rules are comprised of two parts:

1. The matching rule
2. The rule action

The matching rule is what's used to match a NID or a network. The action is what's applied when the rule is matched.

A rule can be uniquely identified using the matching rule or an internal ID which assigned by the LNet module when a rule is added and returned to the user space when they are returned as a result of a show command.

cfg-100, cfg-105, cfg-110, cfg-115, cfg-120, cfg-125, cfg-130, cfg-135, cfg-140, cfg-160, cfg-165

### `lnetctl` Interface

```
# Adding a network priority rule. If the NI under the network doesn't have
# an explicit priority set, it'll inherit the network priority:
lnetctl > selection net [add | del | show] -h
Usage: selection net add --net <network name> --priority <priority>

WHERE:

selection net add: add a selection rule based on the network priority
        --net: network string (e.g. o2ib or o2ib* or o2ib[1,2])
                --priority: Rule priority

Usage: selection net del --net <network name> [--id <rule id>]

WHERE:

selection net del: delete a selection rule given the network patter or the id. If both
                                  are provided they need to match or an error is
returned.
        --net: network string (e.g. o2ib or o2ib* or o2ib[1,2])
                --id: ID assigned to the rule returned by the show command.

Usage: selection net show [--net <network name>]

WHERE:

selection net show: show selection rules and filter on network name if provided.
        --net: network string (e.g. o2ib or o2ib* or o2ib[1,2])

# Add a NID priority rule. All NIDs added that match this pattern shall be assigned
# the identified priority. When the selection algorithm runs it shall prefer NIDs with
# higher priority.
lnetctl > selection nid [add | del | show] -h
Usage: selection nid add --nid <NID> --priority <priority>

WHERE:

selection nid add: add a selection rule based on the nid pattern
                --nid: nid pattern which follows the same syntax as ip2net
                --priority: Rule priority


Usage: selection nid del --nid <NID> [--id <rule id>]

WHERE:

selection nid del: delete a selection rule given the nid patter or the id. If both
                                  are provided they need to match or an error is
returned.
        --nid: nid pattern which follows the same syntax as ip2net
                --id: ID assigned to the rule returned by the show command.
```

```
Usage: selection nid show [--nid <NID>]

WHERE:

selection nid show: show selection rules and filter on NID pattern if provided.
        --nid: nid pattern which follows the same syntax as ip2net
# Adding point to point rule. This creates an association between a local NI and a
remote
# NID, and assigns a priority to this relationship so that it's preferred when
selecting a pathway..
lnetctl > selection peer [add | del | show] -h
Usage: selection peer add --local <NID> --remote <NID> --priority <priority>

WHERE:

selection peer add: add a selection rule based on local to remote pathway
                 --local: nid pattern which follows the same syntax as ip2net
                 --remote: nid pattern which follows the same syntax as ip2net
                 --priority: Rule priority

Usage: selection peer del --local <NID> --remote <NID> --id <ID>

WHERE:

selection peer del: delete a selection rule based on local to remote NID pattern or id
                 --local: nid pattern which follows the same syntax as ip2net
                 --remote: nid pattern which follows the same syntax as ip2net
                 --id: ID of the rule as provided by the show command.

Usage: selection peer show [--local <NID>] [--remote <NID>]

WHERE:

selection peer show: show selection rules and filter on NID patterns if provided.
                 --local: nid pattern which follows the same syntax as ip2net
                 --remote: nid pattern which follows the same syntax as ip2net

# the output will be of the same YAML format as the input described below.
```

## YAML Syntax

Each selection rule will translate into a separate IOCLT to the kernel.

```
# Configuring Network rules
selection:
        - type: net
      net: <net name or pattern. e.g. o2ib1, o2ib*, o2ib[1,2]>
           priority: <Unsigned integer where 0 is the highest priority>

# Configuring NID rules:
selection:
        - type: nid
          nid: <a NID pattern as described in the Lustre Manual ip2net syntax>
          priority: <Unsigned integer where 0 is the highest priority>

# Configuring Point-to-Point rules.
selection:
        - type: peer
          local: <a NID pattern as described in the Lustre Manual ip2net syntax>
          remote: <a NID pattern as described in the Lustre Manual ip2net syntax>
          priority: <Unsigned integer where 0 is the highest priority>

# to delete the rules, there are two options:
# 1. Whenever a rule is added it will be assigned a unique ID. Show command will
display the
#    unique ID. The unique ID must be explicitly identified in the delete command.
# 2. The rule is matched in the kernel based on the matching rule, unique identifier.
#    This means that there can not exist two rules that have the exact matching
criteria
```

```
# Both options shall be supported.
```

### Flattening rules

Rules will have a serialize and deserialize APIs. The serialize API will flatten the rules into a contiguous buffer that will be sent to the kernel. On the kernel side the rules will be deserialzed to be stored and queried. When the userspace queries the rules, the rules are serialized and sent up to user space, which deserializes it and prints it in a YAML format.

## DLC API

```
/* This is a common structure which describes an expression */
struct lnet_match_expr {
        __u32    lme_start;
        __u32    lme_end;
        __u32    lme_incr;
        char     lme_r_expr[0];
};

struct lnet_selection_descriptor {
        enum selection_type lsd_type;
        char                                *lsd_pattern1;
        char                                *lsd_pattern2;

        union {
                __u32                       lsda_priority;
        } lsd_action_u;
};

/*
 * lustre_lnet_add_selection
 *   Delete the peer NIDs. If all peer NIDs of a peer are deleted
 *   then the peer is deleted
 *
 *   selection - describes the selection policy rule
 *   seq_no - sequence number of the command
 *   err_rc - YAML structure of the resultant return code
 */
int lustre_lnet_add_selection(struct selection_descriptor *selection, int seq_no,
struct cYAML **er_rc);
```

## DLC API Structures

```
Defined below
```

# Kernel Space

## Threading model

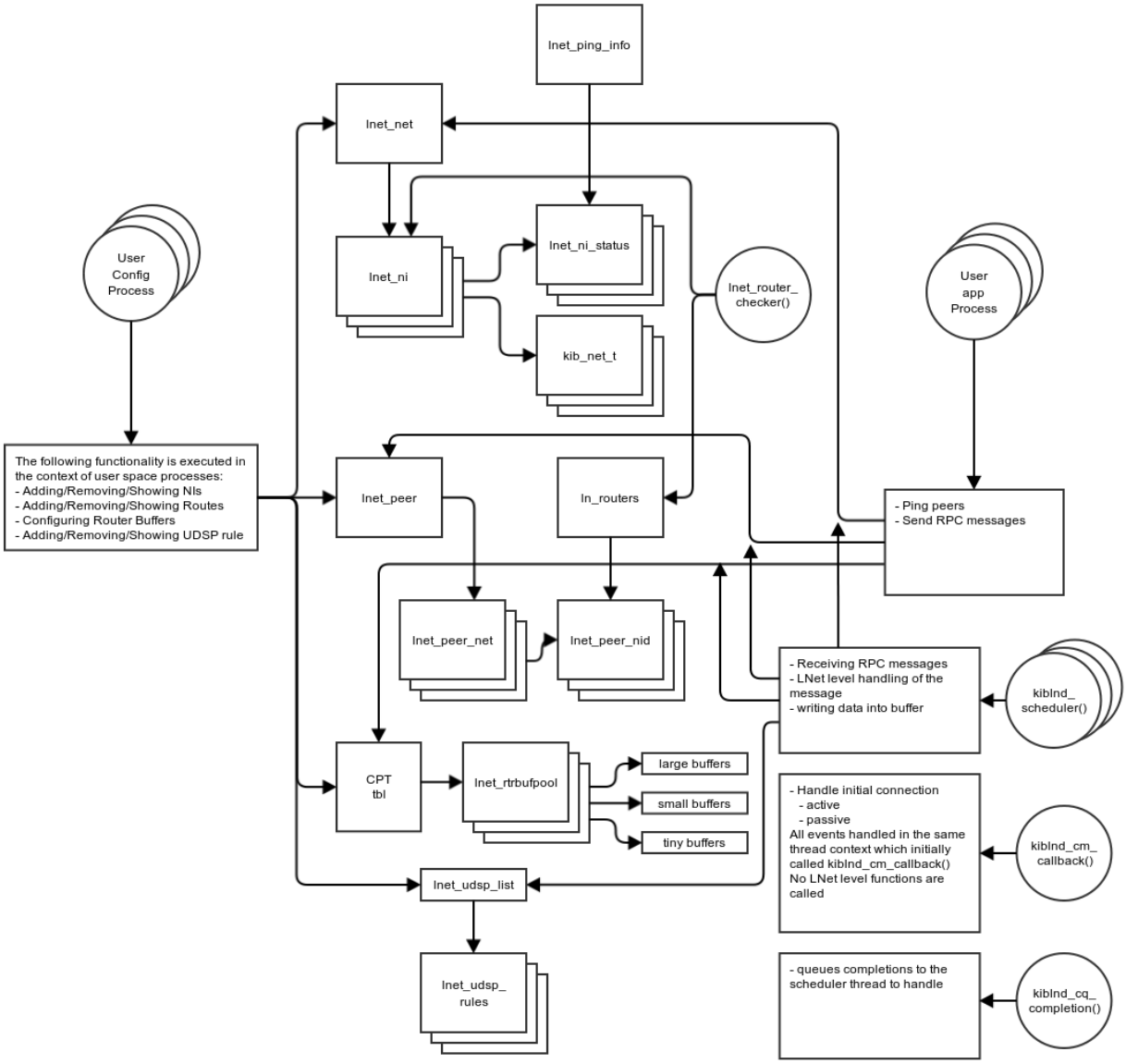



**Figure 3: LNet Threading Model**

## Description

Multi-Rail does not change the LNet threading model and the locking will remain largely the same. However it changes the structures which are accessed from the different thread contexts.

## Locking

The current code adds peers opportunistically, the first time a message is sent to or received from a peer. The peer table is split over the CPTs allowed to lnet, and a peer's NID is used to pick a specific CPT. There is a per-CPT set of spinlocks `(lnet_net_lock/lnet_net_unlock)`, and the spinlock for a CPT must be held when the peer table of that CPT is traversed or modified. Note that the current code does not create peer structs for peers not connected to a local network.

The ioctls that modify or query the LNet configuration use the `ln_api_mutex` in the `the_lnet` for serialization.

# Extending NUMA awareness

Lustre depends on the CPT mechanism of the libcfs kernel module to provide it with information on system topology. In the long run, these interfaces may be replaced with the Linux-native NUMA interfaces, but doing so is far outside the scope of this project. Instead the CPT mechanism will be extended.

## NUMA distance

The concept we need to capture is NUMA distance, which is a measure of the cost of a CPU in one node accessing memory in another. The native interface for this is `node_distance()`. On an x86-64 machine the distance reported by `node_distance()` is typically derived from information provided by the BIOS. Accessing memory on the same NUMA node has some cost, and the reported distance of a node to itself is larger than 0. The value returned by `node_distance()` is a positive integer, larger means larger distance. The distance values have no assigned meaning beyond the ability to compare them.

When all CPTs are entirely restricted to a single node, the distance reported between CPTs is the same as the distance between the nodes that the CPTs live on. But a CPT can span multiple nodes, which raises the question what the distance should be in that case. The options are: minimum, average, and maximum of the distances between the nodes in the CPTs. Using the minimum understates the distance. Using the average depends on addition and division of numbers with no assigned meaning to yield a meaningful result. Therefore we'll use the maximum of the distances, which also has the advantage of being comparatively simple to calculate.

## New CPT Interfaces

The proposal is to add the following functions to the CPT subsystem:

- `cfs_cpt_distance(struct cfs_cpt_table *cptab, int cpt1, int cpt2)` returns the distance between `cpt1` and `cpt2`. If either of `cpt1` or `cpt2` is `CFS_CPT_ANY` then the largest distance in the system is returned – this is consistent with using the maximum distance when a CPT spans multiple nodes.
- `cfs_cpt_of_node(struct cfs_cpt_table *cptab, int node)` returns the CPT that contains `node`. If multiple CPTs contain CPUs from the same node the same CPT number will be returned each time.

The implementation is to add a distance table to `struct cfs_cpt_table` and populate this when the CPT table is created. For debugging purposes `/proc/sys/lnet/cpu_partition_distance` pseudo-file reports the content of the distance table in human-readable form.

More interfaces will be added if we find we need them.

### Memory Descriptors

LNet uses a *Memory Descriptor* (MD) to describe the buffers used by `LNetGet()` and `LNetPut()`. An MD is built by specifying the parameters in a `lnet_md_t`, then calling `LNetMDAttach()` or `LNetMDBind()` to create the internal `struct lnet_libmd`. A CPT number is encoded in the handle that identifies an MD. This CPT is chosen through the call to `lnet_md_link()` in `LNetMDAttach()` or `LNetMDBind()`. The present code works as follows:

- `LNetMDAttach()` derives the CPT from the *Match Entry* (ME) handle passed in. The ME in turn derives its CPT from the match table for the portal.
    - For a *wildcard* portal, `LNET_INS_LOCAL` picks the CPT from the current thread, otherwise the portal number is used.
    - For a *unique* portal, the NID of the peer is used using `lnet_cpt_of_nid()` to match the NID to a CPT.
        - `lnet_cpt_of_nid()` in turn uses `lnet_nid_cpt_hash()` to reduce a NID to a valid CPT number.
        - When a CPT list has been specified for a NI, the CPT is chosen from that list.
- `LNetMDBind()` picks the CPT from the current thread.

We can either add an explicit CPT field to `lnet_md_t` and `struct lnet_libmd`, or build on the existing CPT-aware interfaces and modify how they pick the CPT to better match our requirements.

# Primary NIDs

The assumption that a peer can be identified by a single, unique, NID is deeply embedded in parts of the code. Unfortunately these include the public interfaces of LNet.

- *match entries* (`struct lnet_match_info`) have the peer's NID is one of the possible match criteria.
- *events* (`lnet_event_t`), identify the *initiator* peer by its NID.

For match entries we will translate from the source NID to the primary NID prior to checking for a match. There is an exception in early Discovery because then the primary NID of the peer is not yet known. However, this case is completely contained within LNet.

For events, LNet will provide the primary NID in the `initiator` field. Event handlers may also need the actual source NID so a `source` field will be added to `lnet_event_t`.

The primary user of LNet in the Lustre code is PtlRPC and the OBD and LDLM layer built on top of that which are strongly intertwined with PtlRPC. (Both of these peek into PtlRPC data structures.)

- The `c_peer` field of `struct ptlrpc_connection` identifies the peer by a NID.
- The `rq_peer` field of `struct ptlrpc_request` identifies the peer by a NID.

The `rq_peer` field is set to the primary NID. Since we want PtlRPC to be able to route responses to a specific source NID, a new field, `rq_source` is added for that purpose.

`ptlrpc_uuid_to_peer()` may need to be changed to map the selected peer NID to the primary NID of that peer.

`target_handle_connect()` is a place outside PtlRPC that peeks into PtlRPC datastructures to find a peer's NID. Setting `rq_peer` to the primary NID should suffice.

`ldml_flock_deadlock()` looks at `c_peer` when doing deadlock detection.

# IOCTL Handling

## Adding NI

Handling of the new ADD_NI IOCTL will be done in the `module.c:lnet_ioctl()`

There will not be any parsing required, as all the string parsing will be done in user space.

```
lnet_add_ni(nid, tunables...)
{
        net = NID2NET(nid);
        /* lnet_find_or_create_net()
         *  if net is not created already create it.
         *  if net was just created run the selection net rules using:
         *  lnet_selection_run_net_rule()
         */
        rc = find_or_create_net(net, &n);
        if (rc != 0)
                return -rc;

        /* make sure that nid doesn't already exist in that net */
        rc = add_ni_2_net(nid, tunables);
        if (rc != 0)
                /* delete net if empty */
                lnet_del_net(net);
                return -rc;
        /* run applicable rules */
        /* lnet_selection_run_nid_rules()
         *  Given the nid of the newly added ni, see if that nid matches any defined
rules and
         *  assign the priority accordingly
         */
        if (lnet_selection_run_nid_rules(ni->nid, &ni->priority))
                /* print an error and increment error counters, but don't fail */
        /* lnet_selection_run_peer_rules()
         *       Given the newly added ni, see if any of the peer rules match the new
NI
         *  and create an association between that ni and any remote peer which
matches
         *  the rule. So if there already exists a rule that matches both this new NI
and
         *  an existing peer then create an association between the pair.
         */
        if (!lnet_selection_run_peer_rules(ni, 0))
                /* print an error and increment error counters, but don't fail */
        /* startup the LND with user specified tunables */
        rc = startup_lndni(ni, tunables...);
        if (rc != 0)
                return -rc;
}
```

## Removing NI

Handling of the new DEL_NI IOCTL will be done in the `module.c:lnet_ioctl()`

There will not be any parsing required, as all the string parsing will be done in user space.

```
lnet_del_ni(nid)
{
        /* 0@<network> basically tells us to delete the entire network and all its NIs
*/
        if (nid == 0@<network>) {
                net = NID2NET(nid);
                if (net is invalid)
                        return -EINVAL;

                /* lnet_dyn_del_ni() will need to be modified to iterate through all
                 * NIs in the net and shutdown each one separately. It will be
appropriately
                 * renamed lnet_dyn_del_nis().
                 * lnet_dyn_del_nis() -> lnet_dyn_del_ni()
                 */
                rc = lnet_dyn_del_nis(net);
                return rc;
        }

        ni = nid_2_ni(nid);

        /* clear any references to peer_nis that might have been set
           while running peer rules */

        rc = lnet_dyn_del_ni(ni);

        /* delete the network if it's empty */
        lnet_del_net(net);
        return rc;
}
```

## Adding Peer NID

```
bool lnet_is_peer_nid_unique(nid)
{
        peer_nis = peer_ni_hash_table[lnet_nid2peerhash(nid)];
        for (peer_ni in peer_nis) {
                if (peer_ni->nid == nid)
                        return false;
        }
        return true;
}

int lnet_peer_add_nid(peer, nid)
{
        net = NULL;
        if ((net = peer->net_array[NID2NET(nid)]) == NULL) {
                LIBCFS_ALLOC(net, sizeof(*net));
                if (net == NULL)
                        return -ENOMEM;
        }

        peer_ni = lnet_peer_create_ni(nid);
        if (peer_ni == NULL)
                return -ENOMEN;

        /* run the nid rules on that nid */
        if (lnet_selection_run_nid_rules(nid, &peer_ni->priority) != 0)
                /* output an error but keep on going */

        if (lnet_selection_run_peer_rules(0, peer_ni) != 0)
```

```
                /* output an error but keep on going */

        list_add_tail(peer_ni->nid_list, net->peer_nid_list);
}

int lnet_add_nid_2_peer(nid_id, nid)
{
        if (nid_id != NULL) {
                peer = lnet_find_peer(nid_id);
                if (peer == NULL)
                        return -EINVAL;
        }

        /* verify that nid being added is unique */
        if (!lnet_is_peer_nid_unique(nid))
                return -EINVAL;

        /* allocate a peer if we couldn't find one using the nid_id provided */
        if (peer == NULL) {
                LIBCFS_ALLOC(peer, sizeof(*peer));
                if (peer == NULL)
                        return -ENOMEM;
                rc = lnet_peer_add_nid(peer, nid_id);
                if (rc != 0) {
                        /* delete the peer that was just created */
                        return -rc;
                }
        }

        rc = lnet_peer_add_nid(peer, nid);
        if (rc != 0)
                return -rc;

        return 0;
}
```

## Removing Peer NID

```
int lnet_del_peer_nid(nid)
{
        /* TODO: map from nid to peer_ni directly */
        peer_nis = peer_ni_hash_table[lnet_nid2peerhash(nid)];
        for (peer_ni in peer_nis) {
                if (peer_ni->nid == nid) {
                        /* cleanup the peer_ni and related info:
                                - any links to other local nis
                           free the peer_ni.
                           If this is the last peer_ni in the peer delete the peer
also
                        */
                        return 0;
                }
                return -EINVAL;
        }
}
```

## User Defined Selection Policies

User-defined selection policy rules will use the same `ip2nets` syntax already described in the manual, with the change defined below. The key difference is that this syntax will be parsed in user space and a structural representation will be passed down to the kernel. The kernel will keep the rules in this structural format and will walk the rule tree when applying them to NIDs and local NIs being added or discovered.

```
<ip2nets> :== <net-match> [ <comment> ] { <net-sep> <net-match> }
<net-match> :== [ <w> ] <net-spec> <w> <ip-range> { <w> <ip-range> } [ <w> ]
<net-spec> :== <network> [ "(" <iface-list> ")" ]
<network> :== <nettype> [ <number> ]
<nettype> :== "tcp" | "gni" | "openib" | ...
<ip-range> :== <r-expr> "." <r-expr> "." <r-expr> "." <r-expr>
<r-expr> :== <number> | "*" | "[" <r-list> "]"
# this allows the interface to define a set of CPTs to be associated with.
<iface-list> :== <interface> ["[" <r-expr> "]" "," <iface-list> ]
<r-list> :== <range> [ "," <r-list> ]
<range> :== <number> [ "-" <number> [ "/" <number> ] ]
<comment :== "#" { <non-net-sep-chars> }
<net-sep> :== ";" | "\n"
<w> :== <whitespace-chars> { <whitespace-chars> }
```

## Structure Representation

The syntax defined above will be parsed into an intermediary form, that will be passed to the kernel. The intermediary form is described diagramatically below.

The diagrams below describe the <r-expr> form. The <r-expr> can be used when defining a network, inteface and an IP.

Example:

```
o2ib0, o2ib*, o2ib[1,2], o2ib[1-10/2], o2ib[1-10/2, 13, 14]
```

or

```
192.168.0.[1-10/2, 13, 14]@nettype
# Refer to Lustre Manual for more examples
```

or

```
eth[1,2,3], eth[1-4/2]
```

**Expression Structural Form** **Description**

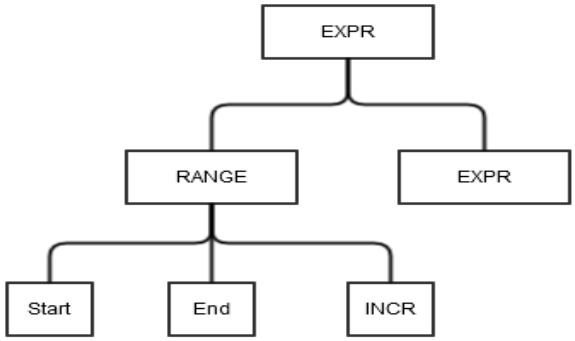



**Figure 4: syntax descriptor**

An expression can be a number:

```
[<num>, <expr>]
represented as:
start == end == NUM
```

An express can be a wild card

```
[*, <expr>]
represented as:
start == 0
end == U32_MAX
INCR == 1
```

An expression can be a range

```
[<start> - <end>, <expr>]
represented as:
start == START_NUM
end == END_NUM
INCR == 1
```

An expression can be a range and an increment

```
[<num-start> - <num-end>/<incr>,
<expr>]
represented as:
start == START_NUM
end == END_NUM
INCR == INCREMENT VALUE
```

When passing the built structural format to the kernel it will need to be serialized, in order not to pass pointers between user space and kernel space.

```
/* The following structures are used to transmit a structural expression
 * to the kernel in flattened form */

struct lnet_offset_descriptor {
        __u32 lodesc_size
        __u32 lodesc_offset;
};

/* address descriptor. Addresses depend on the LND type. Gemini uses hex
 * while IB and TCP use IP addresses. In case of other NIDs which do not
 * use dotted quads, but use only one integer, the below structure can
 * also be used to represent these NIDs. The code to handle the NIDs will
 * take into account the type of the LND and will handle using the below
 * structure appropriately. To isolate these changes, we will consider adding
 * LND level callbacks to handle NID specific operations, to keep LNet LND
 * agnostic. */
struct lnet_address_descriptor {
        struct lnet_offset_descriptor lad_octets[4];
};

struct lnet_ioctl_selection_net_rule {
        char
        lisn_net_name[LNET_MAX_STR];
        __u32                           lisn_priority;
        struct lnet_offset_descriptor   lisn_net_descr;
};
```

```
struct lnet_ioctl_selection_nid_rule {
        char
        lisn_net_name[LNET_MAX_STR];
          __u32                   lisn_priority;
        struct lnet_ip_descriptor lisn_ip_descr;
};

struct lnet_ioctl_selection_peer_rule {
        char
        lisp_net_name[2][LNET_MAX_STR];
    __u32                       lisp_priority;
        struct lnet_ip_descriptor lisp_nid_descrs[2];
};

enum lnet_selection_rule_type {
        LNET_SELECTION_NET_RULE = 0,
        LNET_SELECTION_NID_RULE,
        LNET_SELECTION_PEER_RULE,
};

struct lnet_ioctl_selection_rule {
        enum lnet_selection_rule_type     lisr_selection_type;
        union {
                struct lnet_ioctl_selection_net_rule lisr_net_rule;
                struct lnet_ioctl_selection_nid_rule lisr_nid_rule;
                struct lnet_ioctl_selection_peer_rule lisr_peer_rule;
        } lisr_u;

        char
        lisr_selection_bulk[0];
}

/* These structures are used to store rules internal to the kernel */
struct lnet_selection_net_rule {
        char
        lsnr_net_name[LNET_MAX_STR];
        __u32                             lsnr_net_priority;
        struct lnet_selection_match_expr *lsrn_net_expr;
}

/* for simplicity the nid address will not allow expressions in the network part of
the NID */
/* <expr>.<expr>.<expr>.<expr>@network */
struct lnet_selection_nid_addr {
        char
        lsna_net_name[LNET_MAX_STR];
        struct lnet_selection_match_expr *lsna_octets[4];
};

struct lnet_selection_nid_rule {
        __u32                              lsnr_priority;
        struct lnet_selection_nid_addr    *lsnr_nid_addr;
};

struct lnet_selection_peer_rule {
        __u32                              lsnp_priority;
        struct lnet_selection_nid_addr    *lsnp_nid_addr[2];
};

struct lnet_selection_rule {
        struct list_head                  lsr_list;
        enum lnet_selection_rule_type     lsr_rule_type;
        union {
                struct lnet_selection_net_rule *lsr_net_rule;
                struct lnet_selection_nid_rule *lsr_nid_rule;
                struct lnet_selection_peer_rule *lsr_peer_rule;
        } lsr_u;
};

/* The following APIs add the rules */
```

```
int lnet_selection_add_net_rule(char *net_name, struct lnet_selection_match_expr
*expr, __u32 priority);
int lnet_selection_add_nid_rule(struct lnet_selection_nid_rule *nid_rule, __u32
priority);
int lnet_selection_add_peer_rule(struct lnet_selection_nid_rule *nid_1_rule,
                                                        struct lnet_selection_nid_rule
*nid_2_rule,
                                                        __u32 priority);

int lnet_selection_add_rule(struct lnet_ioctl_selection_rule *selection_rule)
{
        switch (selection_rule->lisr_selection_type):
        LNET_SELECTION_NET_RULE:
                rc = lnet_selection_expand_net_expr(
                        &selection_rule->lisr_u.lisr_net_rule,
                        selection_rule->lisr_bulk,
                        &expr);
                /* check rc */
                lnet_selection_add_net_rule(
                        selection_rule->lisr_u.lisr_net_rule.lisn_net_name,
                        selection_rule->lisr_u.lisr_net_rule.lisn_priority,
                        expr);
                /* check rc */
                break;
        LNET_SELECTION_NID_RULE:
                rc = lnet_selection_expand_nid_expr(
                        &selection_rule->lisr_u.lisr_nid_rule,
                        selection_rule->lisr_selection_bulk,
                        &nid_expr);
                /* check rc */
                lnet_selection_add_nid_rule(
                        selection_rule->lisr_u.lisr_nid_rule.lisn_priority,
                        nid_expr);
                /* check rc */
                break;
        LNET_SELECTION_PEER_RULE:
                rc = lnet_selection_expand_peer_expr(
                        &selection_rule->lisr_u.lisr_peer_rule,
                        selection_rule->lisr_bulk,
                        &nid1_expr, &nid2_expr);
                /* check rc */
                lnet_selection_add_peer_rule(
                        selection_rule->lisr_u.lisr_peer_rule.lisp_priority,
                        nid1_expr, nid2_expr);
                /* check rc */
                break;
        default:
                break;
}

/*
 * lnet_selection_run_peer_rules
 *      Run the peer rules until one matches and stop.
 *  Given a local_ni and a peer_ni, walk the peer rules and try to find a rule which
 *  matches both local_ni and peer_ni nids. Stop on the first found rule.
 *  When match is found, assign a pointer to the peer_ni in the local_ni and vice
versa.
 *  Whenever sending from that local_ni, that peer_ni is used, unless it's down.
 *
 *              local_ni - local ni to match
 *              peer_ni - peer ni to match.
 *  Return 0 on success or an appropriate -error on failure.
 */
int lnet_selection_run_peer_rules(struct lnet_ni *local_ni, struct lnet_peer_ni
*peer_ni);

/*
 * lnet_selection_run_nid_rules
 *      Run the nid rules until one matches and stop.
 *      Given a nid, walk the nid rules and try to find a rule which matches
 *      Stop at the first one found. When match is found assign the priority value
 *  to the OUT parameter.
```

```
 *
 *               nid - nid to match
 *               priority [OUT] - priority to assign the nid
 *
 * Return 0 on success or an appropriate -error on failure.
 */
int lnet_selection_run_nid_rules(lnet_nid_t nid, __u32 *priority);

/*
 * lnet_selection_run_net_rules
 *       Run the net rules until one matches and stop.
 *       Given a net, walk the net rules and try to find a rule which matches.
 *       Sop at the first one found. When match is found assign the priority to the
 *       net->priority
 *
 *               net - net to match
 *
 *       Return 0 on success or an appropriate -error on failure.
 */
int lnet_selection_run_net_rules(struct lnet_net *net);

/* this functions are called on every local_ni, peer_ni or net created */
```

# Dynamic Behavior

## Overview

Dynamic behavior is mainly concerned with the following:

- Sending messages `(LNetPut(), LNetGet())`
- Receiving messages `(lnet_parse())`
- Dynamic Peer Discovery ("Discovery" for short).

## Sending Messages

The entry points into LNet are via the APIs:

- `LNetPut()` - initiate an asynchronous PUT operation
- `LNetGet()` - initiate an asynchronous GET operation

An LNet Put operation consist of an `LNET_MSG_PUT` message with the payload, and an `LNET_MSG_ACK` that confirms receipt of the payload. The caller of `LNetPut()` can indicate that it doesn't need an ACK to be sent.

An LNet Get operation consists of an `LNET_MSG_GET` message, and an `LNET_MSG_REPLY` that contains the payload or error code.

At the LNet level, we'll be talking about *message/reply* pairs: message being `PUT` or `GET`, reply being `ACK` or `REPLY`. An LNet reply is sent to the same NID that the message was sent from: the way interface credits are managed requires this. This means that the NI selection algorithm cares whether we are looking at a message or a reply, and the algorithm must be bypassed in the latter case.

The RPCs used by PtlRPC are built on top of `LNetGet()` and `LNetPut()` calls. An RPC consists of a *request* and a *response*. A request is typically a Put, and the response is another Put, which may then trigger a Get to pull additional data from the remote node. A PtlRPC response can be sent to a different

NID than where the request was sent from, though this is usually only desirable if there some problem sending to the original NID.

The callers of `LNetGet()` and `LNetPut()` need to be reviewed. The interpretation of the *self* parameter will be somewhat different, and the difference matters. The distinction is between sending from any available NI to any available peer NI, versus a strong preference for a particular NI/peer NI pair.

Sending a message may trigger Discovery.

## NUMA Awareness

NUMA information needs to be provided by the higher level layers when calling `LNetPut()` and `LNetGet()`. This NUMA information is then used by selection API to determine the optimal local_ni and peer_ni pair to use for sending a message.

To avoid altering these APIs, since they are used by modules outside of Lustre, the NUMA information will be added to `struct lnet_libmd`. See Memory Descriptors above for a discussion of this point.

Since the md is attached to the msg and the msg is already part of the `lnet_send()` parameters, there will be no need to modify the `lnet_send()` API.

The logic of the `lnet_send()` API will need to change however.

In summary, `lnet_send()`, given a destination NID (in the msg) and a src_ni (or `LNET_NID_ANY`), must determine the best local NI to use based on the NUMA criteria in the MD and the best destination NID to use.

The pseudo code below describes the algorithm in more details. snd-005, snd-010, snd-020, snd-030, snd-035, snd-040, snd-045, snd-050, snd-055, snd-060, snd-065, snd-070, snd-075

snd-015 - NUMA APIs were added in some form, at least since 2.6.1**;** and therefore will pose no problems for this project.

## Resending Messages

LNet has been designed on the assumption that a message is sent once, and failure is reported either immediately via the return value of `LNetGet()` or `LNetPut()`, or later via the status reported in the `LNET_EVENT_SEND` event. When this event is posted the memory used for the message can be reused. We have little choice but to rely on the LND to tell LNet whether a message was successfully sent. Detecting send failure will therefore be best-effort.

Any attempt to resend a message needs to hook into `lnet_finalize()`, which is the function that releases the buffers and posts `LNET_EVENT_SEND`. The simplest approach is to modify this function so that on error it doesn't release the buffers nor posts the event, but instead initiates resending the message.

This can be extended by adding a timeout to a message being sent, and then initiate a retry if the timeout expires before `LNET_EVENT_SEND` has been posted. Now the same message may be successfully sent multiple times if there is some network delay. `lnet_finalize()` must track whether another attempt to send this message is still in progress. This is in addition to the code initating resending a message on a failure signaled by the LND. Note that interface and peer credits cannot be released until after the LND

has signaled a failure by calling `lnet_finalize()`, and `LNET_EVENT_SEND` cannot be posted until all concurrent attempts to send a message have been resolved. Progress is limited by the slowest success or failure. This makes the value of the extra complexity involved somewhat doubtful.

## Local NI Health

A local NI can be marked bad if the LND signals a failure of the interface. This would be a hard failure. A timer can be used to re-check periodically – this is something Fujitsu implemented for the `o2ib` LND and worth copying.

We can also mark it unhealthy if attempts to send messages through this NI fail, especially if the failures exceed some set rate. One method is to keep a decaying sum of soft failures per NI, and comparing the sums for each NI when selecting which local NI to use.

## Peer NI Health

A peer NI can be marked unhealthy when we see failures when sending a message to that peer NI. For apparent soft failures this can be rate based, and a decaying sum of failures could be used to select between different peer NIs.

For apparent hard failures it is worth noting that PING/PUSH information contains the status of each interface. This is a mechanism by which presence of and recovery from hard failures can be communicated. Rather than have a peer actively push such information, it is likely better to have nodes pull it when they need it. Such a pull (done by pinging the peer, of course) can be done occasionally as long as other, healthy, peer NIs are available.

## Selection Algorithm Pseudo-code

The following pseudo-code illustrates how a local NI, peer NI pair can be selected in a reasonably efficient manner.

```
# Find the peer via its nid.
peer_ni = lookup(peernid);
peer = peer_ni->peer_net->peer;

# Keep track of the best selection so far.
best_peer_net = NULL;
best_ni = NULL;
best_peer_ni = NULL;
best_gw = NULL;

# Keep track of the best selection criteria seen
best_net_priority = LOWEST_PRIORITY;
best_numadist = maxnumadist(cptab); # Worst in system.
preferred = false;
best_peer_credits = INT_MIN;
best_credits = INT_MIN;

# Find a ni by walking the peer's peer_net_list,
# then walking the releated net's ni_list.
for (peer_net in peer->peer_net_list) {
        if (peer_net_not_connected(peer_net))
                continue;
        # If all peer_ni on this peer_net are unhealthy,
        # then the peer_net itself is marked unhealthy.
        if (peer_net_not_healthy(peer_net))
                continue;
        # Smaller priority value means higher priority
        # Lower-priority networks can be skipped if a viable
```

```
# network has been found. If the peer_net_list is
# sorted by priority we can break out of the loop here.
if (best_peer_net && best_net_priority < peer_net->priority)
        continue;
# Candidate peer_net, look at each ni connecting to it
net = peer_net->net;
# The assumption is that a network is either direct-connected
# or routed, but never both. Note that you can give a direct
# connected network a lower (network) priority than a routed
# network, in which case the routed network will be preferred.
net_gw = NULL;
if (net_is_routed(net)) {
        # Look for a suitable gateway.
        # As written this combines lnet_peer_ni with lnet_route
        for (gw in net->gateway_list) {
                if (gw_not_healthy(gw))
                        continue;
                if (!net_gw) {
                        net_gw = gw;
                        continue;
                }
                if (net_gw->priority < gw->priority)
                        continue;
                if (net_gw->hops < gw->hops)
                        continue;
                if (net_gw->txqnob < gw->txqnob)
                        continue;
                if (net_gw->txcredits > gw->txcredits)
                        continue;
                # The seq is the final tiebreaker
                if (net_gw->seq - gw->seq <= 0)
                        continue;
                # Bump seq so that next time the tie breaks
                # the other way
                net_gw->seq = gw->seq + 1;
                net_gw = gw;
        }
        # No gateway, no route
        if (!net_gw)
                continue;
        # At least as good as the globally best gw?
        if (best_gw) {
                if (best_gw->priority < net_gw->priority)
                        continue;
                if (best_gw->hops < net_gw->hops)
                        continue;
                if (best_gw->txqnob < net_gw->txqnob)
                        continue;
                if (best_gw->txcredits > net_gw->txcredits)
                        continue;
        }
        # Local connected net for gw
        net = net_gw->net;
}
# Look for ni on net
for (ni in net->ni_list) {
        if (ni_not_healthy(ni))
                continue;
        # NUMA distance between ni and md (and current
        # thread), larger is worse.
        dist = numadistance(cptab, ni, md);
        if (dist > best_numadist)
                continue;
        # Select on NUMA distance, then local credits
        # Negative credits imply queued messages
        # A sequence number as a final tiebreaker/load
        # spreader
        if (dist < best_numadist) {
                best_numadist = dist;
        } else if (ni->credits <= best_credits) {
                continue;
        } else if (best_ni) {
```

```
                        if (best_ni->seq - ni->seq <= 0)
                                continue;
                        best_ni->seq = ni->seq + 1;
                }
                best_peer_net = peer_net;
                best_net_priority = peer_net->priority;
                best_ni = ni;
                best_credits = ni->credits;
                best_gw = net_gw;
        }
}
# If there is no best_ni we've failed.
if (!best_ni)
        failure;
# Look for a peer_ni connected to the best_ni by walking
# the peer_ni list of the best_peer_net.
for (peer_ni in best_peer_net->peer_ni_list) {
        if (peer_ni_not_healthy(peer_ni))
                continue;
        # Is the best_ni a preferred ni of this peer_ni?
        ni_is_pref = (best_ni in peer_ni->preferred_ni_set);
        # If no preferred ni has been seen yet, and this ni
        # is preferred by this peer_ni, pick this peer_ni.
        # If a preferred ni has been seen, and this ni is
        # not preferred by this peer_ni, skip it.
        # Otherwise, select on available peer_credits.
        # Finally, a sequence number to rotate load
        if (!preferred && ni_is_pref) {
                preferred = true;
        } else if (preferred && !ni_is_pref) {
                continue;
        } else if (peer_ni->peer_credits <= best_peer_credits) {
                continue;
        } else if (best_peer_ni) {
                if (best_peer_ni->seq - peer_ni->seq <= 0)
                        continue;
                best_peer_ni->seq = peer_ni->seq + 1;
        }
        # We have (new) favorite.
        best_peer_ni = peer_ni;
        best_peer_credits = peer_ni->peer_credits;
}
# No best_peer_ni means we've failed. That should only
# happen if all peer_ni of this peer_net are unhealthy.
# So the peer_net must now be marked unhealthy and the
# selection restarted from the top
if (!best_peer_ni) {
        mark_peer_net_unhealthy(best_peer_net);
        restart_from_top;
}
# Yay!
success(best_ni, best_peer_ni, best_gw);
```

## Receiving Messages

The way LNet processes received messages will remain largely the same except for the modifications that will need to occur when accessing the internal structures, namely `lnet_ni` and `lnet_peer`, as these have changed as described earlier in the document.

Receiving a message may trigger Discovery.

## Backward Compatibility

The features of the existing code noted in the Primary NIDs section imply that a multi-rail capable node should always use the same source NI when sending messages to a non-multi-rail capable node. The

likely symptoms of failing to do this include spurious resets of PtlRPC connections, but also more subtle problems like failures to detect flock deadlocks.

## Dynamic Peer Discovery

Dynamic Peer Discovery ("Discovery" for short) is the process by which a node can discover the network interfaces it can reach a peer on without being pre-configured. This involves sending a ping to the peer. The ping response carries a flag bit to indicate that the peer is multi-rail capable. If it is the node then pushes its own network interface information to the peer. This protocol distributes the network interface information to both nodes and subsequently the nodes can excercise the peer network interfaces as well as its own, as described in further detail in this section. Discovery can be enabled, disabled or in verification mode. If it is in verification mode, then it will cross reference the discovered peer NIDs with the configured NIDs and complain if there is a discrepancy, but will continue to use the configured NIDs. cfg-085, dyn-005, dyn-010, dyn-015, dyn-020, dyn-025, dyn-030, dyn-035, dyn-040, dyn-045, dyn-050, dyn-055, dyn-060, dyn-065

### Discovery handshake

Discovery happens between an active node which takes the lead in the process, and a passive node which responds to messages from the active node. The following diagram illustrates the basic handshake that happens between the active and passive nodes. If the full handshake completes, both nodes have discovered each other.

In addition, the diagram illustrates the cases where either node adds or removes a local NI, and uses a push to inform the other node.

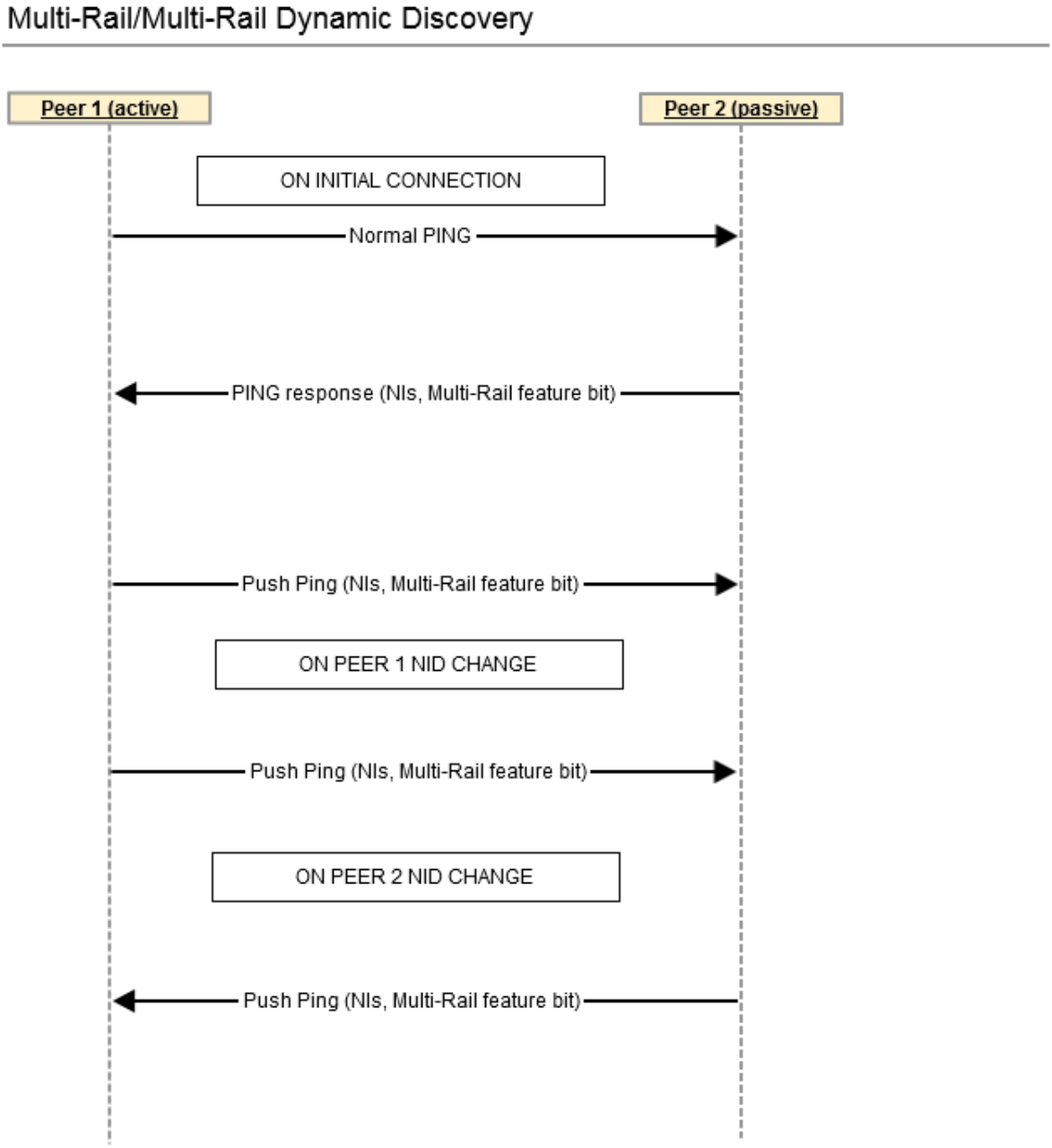



**Figure 5: Dynamic Discovery Overview**

## Discovery thread

To handle some of the work required for discovery, a new kernel thread will be created: the *discovery* thread. The discovery thread is responsible for handling large parts of the active and passive side of discovery. Using a separate thread alleviates some concerns regarding race conditions (but not all) as well as concerns regarding stack depth.

The discovery thread handles:

1. Active side of discovery:
    1. sending the ping message,
    2. parsing the ping reply,
    3. updating datastructures,
    4. sending the push.
2. Passive side of discovery:
    1. parsing the data from a push message,
    2. updating datastructures

The discovery thread is started when LNet initializes, and runs even when discovery is disabled. If discovery is enabled later, which can be done using DLC, this ensures that the peer state is correctly prepared.

## Discovery handshake code flow

The following diagram illustrates the code flow in the active and passive nodes during the discovery handshake. It assumes that this is the first time the active and passive nodes communicate.

In the diagram, *send* is the thread sending a message, *recv* is the helper that handlers an incoming message (managed by the LND), and *discovery* is the discovery thread.

The use of the following locks is illustrated:

- `lnet_net_lock`, a spinlock protecting peer state and lookups
- `lnet_res_lock`, a spinlock protecting memory descriptors and event queues
- `lnet_peer_lock`, new mutex protecting peer configuration

To add a peer to the lookup tables, both the lnet_net_lock and lnet_peer_lock must be held. The lnet_peer_lock exists to ensure mutual exclusion when more complicated peer structures involving multiple peer_ni structures need to be built.

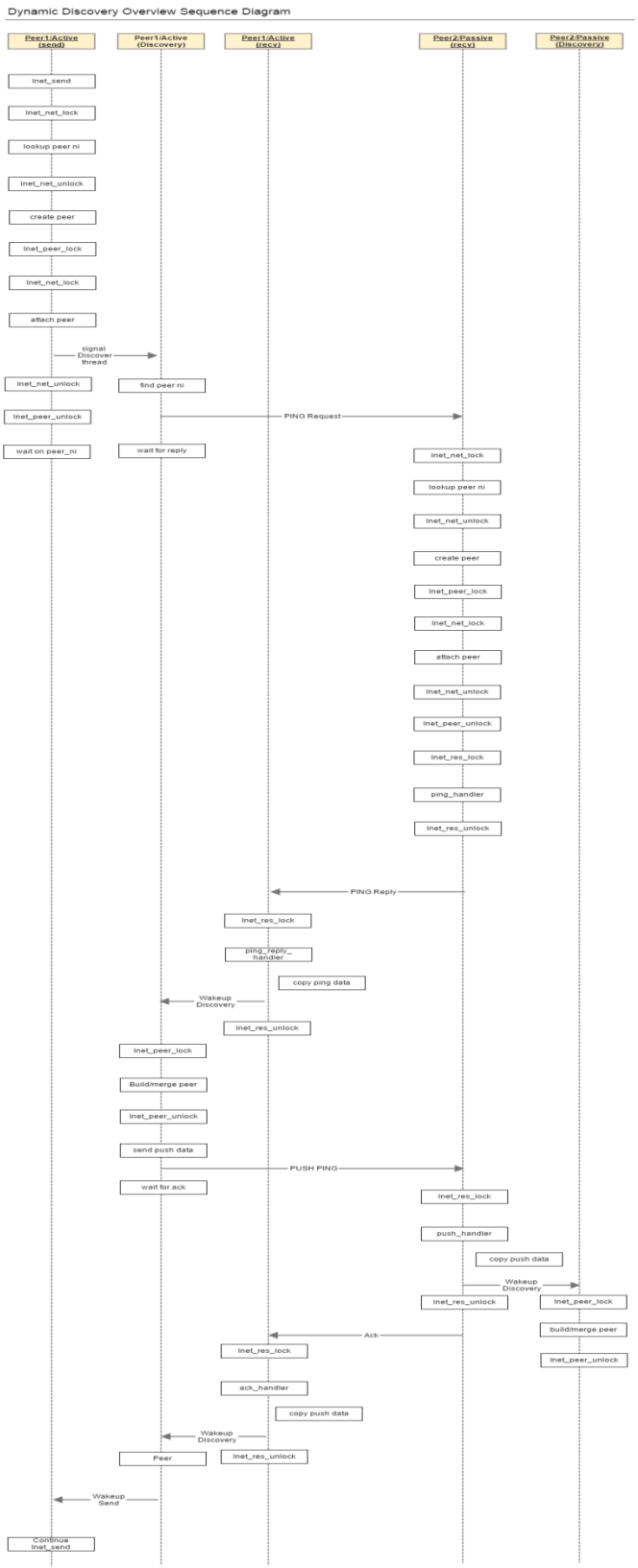



**Figure 6: Dynamic Discovery Detailed Sequence Diagram**

Notes:

- The `ping_handler`, `ping_reply_handler`, `push_handler`, and `push_ack_handler` are event queue callbacks, and called with the `lnet_res_lock` held. This is a spinlock, so any memory allocations done in the handler must be `GFP_ATOMIC`. In general we don't want to do anything long-running here.
- Note that the `lnet_net_lock` nests inside the `lnet_res_lock`.
- If we're re-using the memory in the MD (hard to see how to avoid this) then we have to copy out the incoming ping and push data in `ping_reply_handler` and `push_handler`. This isn't much of an issue for a ping, as we initiated it and can prepare for this. It is trickier for an incoming push, which arrives with little prior warning, beyond the fact that we were just pinged.
- Waiting for the Push Ack ensure there is never more than one push in progress to a single peer NI.
- In the sketch above the Active peer can send its message to the Passive peer before the Passive peer has completed constructing the peer struct.

## Discovery and DLC

Discovery interacts with peer configuration through DLC. Combine these and a peer can be created in four different ways:

1. DLC configuration of a peer.
    1. Discovery can be enabled
    2. Discovery can be disabled
2. DLC/module parameter configuration of a router.
3. In `lnet_send()` when sending a message. (The active side of discovery.)
4. In `lnet_parse()` when parsing a message. (The passive side of discovery.)

Routers are an interesting case: discovery is not exactly *required* for them, because they must be defined through the DLC/module parameter configuration. Without that configuration a node does not know which remote networks can be reached through which routers.

## Discovery race conditions and edge cases

Discovery is initiated by the active side in `lnet_send()`. When the passive side calls `lnet_send()` to send a reply, this does not initiate a Discovery round. For an unknown peer, the passive side will need to set up enough of the `peer_ni`, `peer_net`, and peer datastructures to ensure that a ping reply can be sent. At the point where it does this in `lnet_parse()`, the passive side will not yet know whether the active is multi-rail capable or not.

**Use case scenarios to consider:**

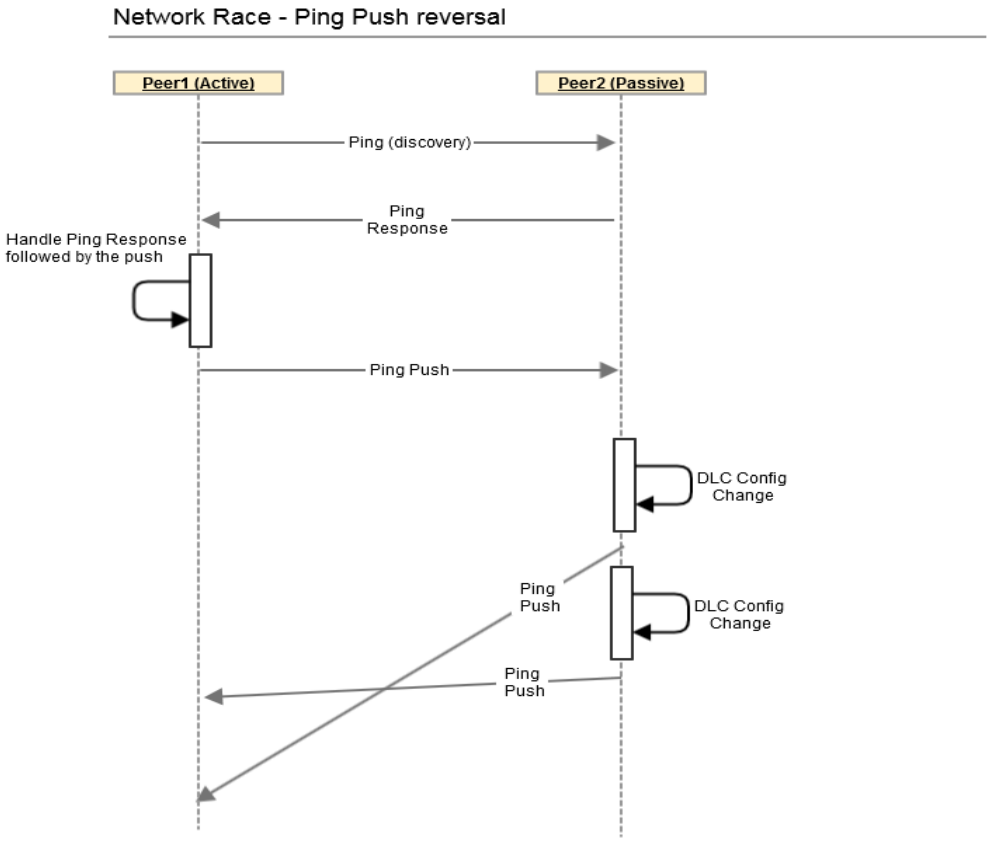
The following list covers various scenarios that were considered in the design of the Discovery Algorithm. The main point is that Discovery is subject to many possible races, which we address by tracking the state of each `peer` and `peer_ni` involved.

1. Note that the passive node sends a ping reply to the active node before the active node knows whether it should do a ping push.
    1. At this point the passive node does not know whether the active node is uprev or downrev.
    2. As noted above, the LNet reply always goes to the originating NID, so the passive node has enough information to be able to send it.

2. The active node can be doing discovery on multiple NIDs of the passive node at the same time.
    1. Active node:
        1. The active node has to create `peer/peer_net/peer_ni` (at the very least `peer_ni`) datastructures to be able to send a ping message
        2. The active node now has muliple `peer/peer_net/peer_ni` structures for the same peer.
        3. On receipt of the ping reply the active node merges these structures.
        4. Having merged these structures, the active node sends a ping push message.
        5. The active node should be smart enough to not send multiple ping push messages in this case.
        6. The serialization we obtain by having a single dynamic discovery thread helps here.
    2. Passive node:
        1. The passive node has to create `peer/peer_net/peer_ni` datastructures to be able to send a ping reply.
        2. At the point where the passive node does this, it doesn't know whether the active node is uprev or downrev.
        3. If downrev, the passive node will not receive further information from the active node.
        4. Therefore the datastructures set up must be complete for a downrev active node.
        5. An uprev active node may have multiple pings in flight to different NIDs, prompting creation of multiple peer structures.
        6. On receipt of the ping push message, these structures must be merged.
        7. Further pushes serve to update these structures.
3. Dynamic discovery and DLC configuration can update the same peer at the same time.
    1. Serialize updates through a *peer* mutex, and protect lookups with per-CPT spinlocks.
    2. A lookup needs just the per-CPT spinlock.
       An update must hold both the mutex and all per-CPT spinlocks – `LNET_LOCK_EX`. It needs this because a single per-CPT lock protects lookups of a peer NID, but also traversal of the `peer_ni` list in the `peer_net`, and the `peer_net_list` in the peer. So all per-CPT locks need to be held if the `peer_ni_list` or `peer_net_list` is to be changed.
4. Can DLC modify discovered peers?
    1. Presumably yes.
    2. Troublesome case is deleting a peer NI that we're just using in discovery.
    3. This is not different from the normal case of trying to delete a peer NI that is currently in use.
    4. The peer NI must be idled first, which implies that the discovery round on that peer NI must be allowed to finish.
    5. Discovery can push a NI list that does not include the NI going idle, even though it uses that NI.
    6. This is similar to the normal case where DLC removes an active NI.
    7. While waiting for a NI to go idle, the peer mutex must be released, to avoid dynamic discovery deadlocking with DLC.
    8. We probably do not want yet another DLC request to come in and try to re-add the peer NI before all the above has finished.
    9. So the peer mutex mentioned above is *not* the `ln_api_mutex` that the ioctls serialize on.
    10. The api mutex must be held by the thread doing the ioctl across the entire operation, to avoid this configuration race.
    11. When both are held, the api must be locked before the peer mutex.

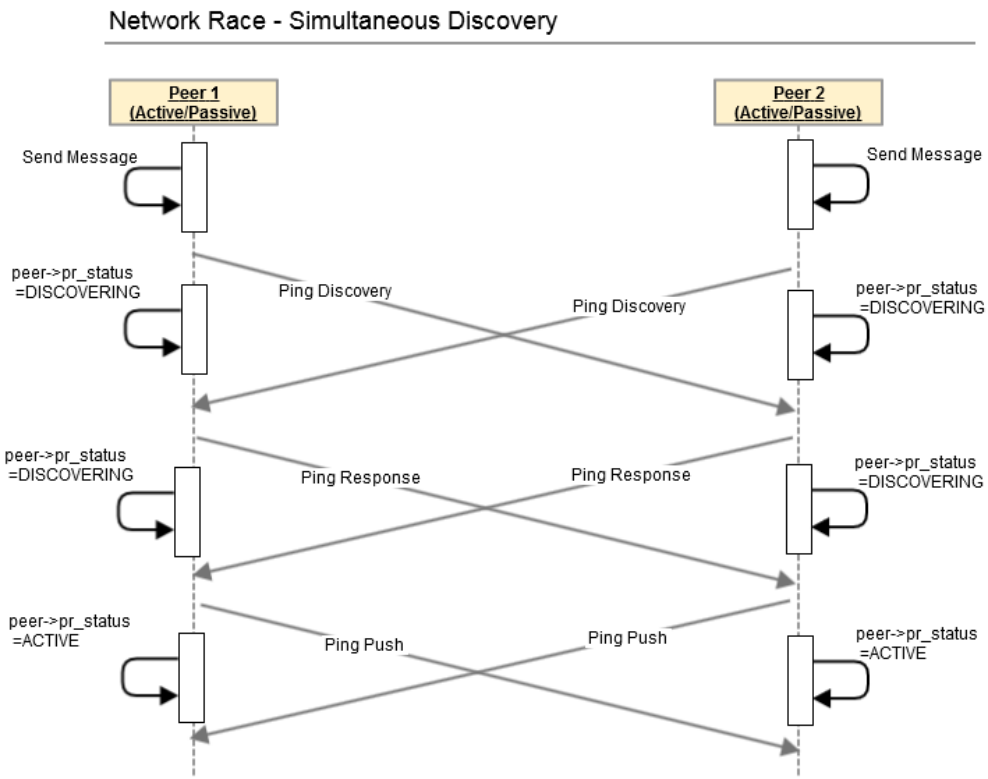
5. Can discovery modify DLC configured peers?
    1. Presumably yes.
    2. When DLC adds a peer NI, it can hold the peer mutex across the entire operation.
    3. When DLC removed a peer NI, it ensured it was idle first.
    4. Discovery always sees a coherent peer datastructure to work on.
6. The active node has discovery enabled, the passive node has discovery disabled.
    1. The passive may not have a configuration for the peer. In a cluster with only a few multi-rail nodes, it is plausible to just not explicitly configure the non-multi-rail peers.
    2. There are three approaches:
        1. the passive node just drops any push on the floor. In this case the dynamic discovery thread need not be running.
        2. the passive node verifies its configuration using the push message received. In this case the dynamic discovery thread needs to be running.
            1. a push containing more than one interface merits a complaint
            2. a push containing a single interface is accepted without complaint
        3. the passive node updates its configuration using the push.
7. The active node has discovery disabled, the passive node has discovery enabled.
    1. The passive node is prompted to create the `peer/peer_net/peer_ni` datastructures as usual
    2. If the active node wasn't DLC configured on the passive node, then the passive node will not detect that the active node is uprev. The relevant ping traffic never happens.
    3. A multi-rail node on which discovery is disabled must be added to the DLC configuration of all its relevant peers.
8. Active side enables dynamic discovery
    1. While dynamic discovery is disabled all peers added via DLC are moved to the ACTIVE state, and no dynamic discovery is performed
    2. When dynamic discovery is enabled peers which are in the ACTIVE state are not dynamically discovered.
        1. The other option is to have it retroactive and go through all the peers and determine if they have been dynamically discovered and if not then initiate dynamic discovery.
            1. This is likely to cause a spike in traffic
            2. In large systems this could cause a heavy load on the nodes since there could be potentially thousands of peers.
    3. Further communication with peers in ACTIVE state does not trigger dynamic discovery
    4. New peers added via DLC are moved to the WAITING-VERIFICATION state and on first message to these peers dynamic discovery is triggered.
    5. If dynamic discovery is disabled any messages sent on peers in the WAITING_VERIFICATION state, will cause the peers to move directly to the ACTIVE state with no discovery round triggered.
    6. Messages sent to peers that do not exist yet in the system trigger dynamic discovery if dynamic discovery is enabled.

9. Network Race: Ping push from peer is flipped. This can happen in both directions. In this case the peer receiving the peer will not be alble to distinguish the order of the push and could end up with outdated information.
    1. To resolve this situation a sequence number can be added in the ping push, allowing the recieving side to determine the order.
    2. This will entail the receiving side to maintain the last sequence number of received push.
    3. If the push received has a sequence number which is greater than what it currently has for that peer, then update, otherwise ignore the push since it's has outdated information.
    4.




1. **Figure 7: Push/Push Race Condition**

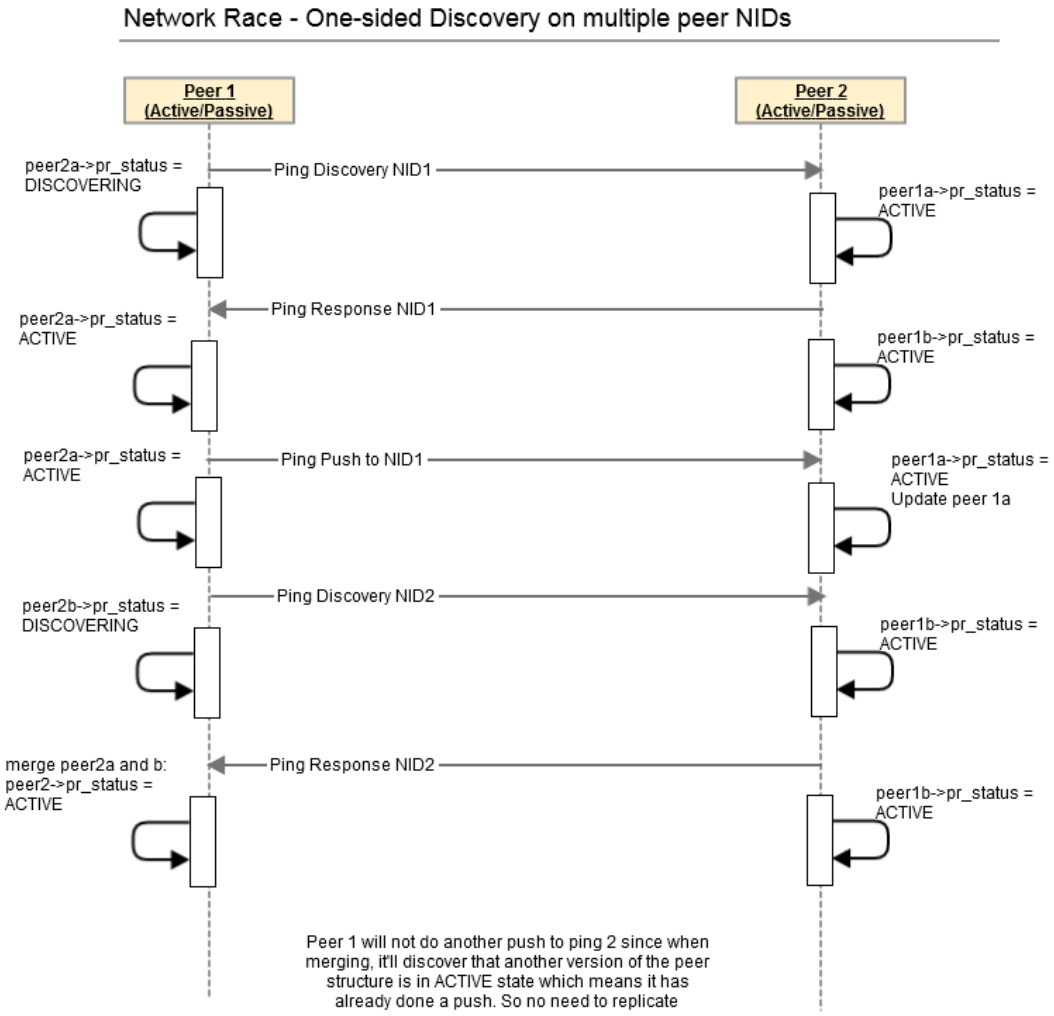
10. In the case when two peers simultaneously attempt to discover each other, each peer will create the corresponding peer/peer_net/peer_ni structures as it would normally do, and will transition its states according to the FSM. This scenario should be handled through the normal code path.
    1.




1. **Figure 8: Simultaneous Discovery Scenario**

2. The following variations could occur
    1. The node can receive a ping and create the corresponding peer structures before it starts peer discovery.
        1. In this case when the message is attempted to be sent to that peer, the structures are found and the peer is going to be in ACTIVE state, and no discovery round will be triggered.
    2. The node can receive a ping after it has started its ping discovery round
        1. In this case the peer structure will be found in the DISCOVERING state. The ping response will be sent back (possibly before the peer state is examined, but it's not important)
    3. the node can receive a ping response before it sends its own ping response. This is the standard case. The ping discovery protocol would be completed at this point
    4. The node can receive a ping push before it has sent it's own ping push. This would result in it updating it's own structures. This is again handled in the normal case.

11. In the case when one node attempts to discover the same peer on multiple NIDs. Multiple `peer/peer_net/peer_ni` structures will be created for each one of the NIDs, since at this point the node doesn't know that it's the same peer. On ping response the node will send a ping push and transition the corresponding peer state to ACTIVE (note the order). When the second ping response on NID2 is received the information in the ping response is used to locate peer2a and the structures are merged. Since the other peer found is already in ACTIVE state, then there is no need to send another ping push. On the passive side, a similar process occurs. If the ping is sent from two different sources, then two `peer/peer_net/peer_ni` structures are created, and then merged when the push is received, which serves to link both structures. If the ping is sent from the same src NID, then the peer is created on the first ping and found on the second ping. No merge is required.
    1.




1. **Figure 9: One-sided Discovery on multiple peer NIDs**

12. It is possible to have simultaneous discovery on multiple NIDs. This is a combination of scenario 10 and 11. The handling of both scenarios apply here.

## The Discovery algorithm

The discovery algorithm is best characterized through the state of the peer as seen from the node doing discovery. To drive discovery we need answers to the following questions for each peer:

1. Have we received the peer's NI information? Two ways to get it:
    1. This node pinged the peer.
    2. The peer pushed this node.
2. Has the peer received the local node's NI information. Again two ways:
    1. This node pushed to the peer.
    2. The peer pinged this node.
3. Has local NI config changed from what the peer was told. Several ways this can happen:
    1. DLC update
    2. Interface hotplug

The state of a peer is a combination of the following bits of information:

- L: Local config sent to peer
- P: Peer config merged
- M: Multi-rail capable peer
- D: Data received from peer, not yet merged
- R: Reply to ping pending
- A: Ack pending
- Q: Queued for the discovery thread to work on
- C: Configured by DLC
- S: Size of MD buffers need to be increased

The following discussion will mention these by referring to the underlined letters: L, P, M, D, R, A, Q, C, S. The discussion mostly treats these as 2-state flags, but the actual implementation may differ. For example, the L state can be implemented by tracking the sequence number of the local NI config sent to the peer, and comparing this to the current sequence number. The states have been defined in such a way that initial state of a `peer` has all of them cleared.

Some of the `peer` states tie into the state of its attached `peer_ni`. The `peer_ni` states mentioned below are:

- PC: This `peer_ni` was configured by DLC.

Where applicable, the description attempts to characterize both the behavior where DLC overrides Discovery, and vice versa.

1. DLC
    1. DLC adds a local NI
        1. clear L on all peers
    2. DLC deletes a local NI
        1. clear L on all peers
    3. DLC adds a peer NI to this peer
        1. clear P
        2. find or create a `peer_ni`
            1. if the `peer_ni` doesn't exist yet, create it
            2. if the `peer_ni` is attached to the `peer` and PC is clear, fine, DLC confirms what discovery already found

            3. if the `peer_ni` is attached to the `peer` and PC is set, we have a duplicate DLC add
            4. if the `peer_ni` is attached to a different `peer` and PC is clear
                1. if P is set on that `peer` complain: DLC conflicts with discovery
                2. if P is clear on that `peer` then there should not be a conflict
            5. if the `peer_ni` is attached to a different `peer` and PC is set, we have conflicting DLC
        3. attach the `peer_ni` and set PC on it
        4. if DLC overrides discovery, all `peer_ni` with PC clear must be detached
        5. set C iff all `peer_ni` have PC set
    4. DLC deletes a peer NI from this peer
        1. clear P
        2. detach the `peer_ni`
            1. if PC is not set on the `peer_ni` then DLC is removing a discovered `peer_ni`, which we might want to complain about
            2. if PC is set on the `peer_ni` then DLC is removing a DLC-configured `peer_ni`
        3. if DLC overrides discovery and C is not set, then all `peer_ni` with PC not set must be detached
        4. set C iff all `peer_ni` have PC set, indicating the `peer` now matches the DLC
2. Forced rediscovery triggered by the upper layers (PtlRPC) if they somehow conclude that the peer config might be incorrect.
    1. if discovery is disabled
        1. if C is set, complain about possibly-bad DLC
        2. if C is not set, complain about possibly-bad previous discovery
    2. if discovery is enabled (note that DLC overrides discovery)
        1. if C is set, complain about possibly-bad DLC
        2. if C is not set, proceed with "if discovery is enabled"
3. Sending a message
    1. if the type is `LNET_MSG_ACK` or `LNET_MSG_REPLY`
        1. continue sending
    2. else if the portal is `LNET_RESERVED_PORTAL`
        1. continue sending
    3. else if L and P are set (`peer` is up-to-date)
        1. continue sending
    4. else if discovery is enabled
        1. set Q
        2. wait for discovery thread to signal us
    5. else (discovery is disabled)
        1. continue sending
4. Ping handling
    1. Sending a Ping message
        1. set R
    2. Receiving a Ping reply (compare with receiving a push)
        1. clear R
        2. set M if the peer is multi-rail
        3. clear M if the peer is not multi-rail
        4. tag any `peer_ni` not mentioned in the data as do-not-use
        5. clear the do-not-use tags from all `peer_ni` mentioned in the data and attached to this `peer`
        6. set D if a copy of the data can be stored
        7. set S if the MD buffers were too small to receive the full data
        8. set Q
    3. Receiving a Ping message (being pinged by a remote node, EQ callback)

            1. set L
            2. send reply (automatic once the callback completes)
5. Push handling
    1. Sending a Push message
        1. set A
    2. Receiving a Push ack (EQ callback)
        1. clear A
        2. set L
        3. clear Q (dequeues, wakes waiters)
    3. Receiving a push message (being pushed by a remote node, EQ callback)
        1. clear P
        2. set M if the peer is multi-rail
        3. clear M if the peer is not multi-rail – this should never happen
        4. tag any `peer_ni` not mentioned in the data as do-not-use
        5. clear the do-not-use tags from all `peer_ni` mentioned in the data and attached to this `peer`
        6. set D if a copy of the data can be stored
        7. set S if the MD buffers were too small to receive the full data
        8. set Q (don't wait, we're in a callback)
        9. send ack (automatic once the callback completes)
6. Merge received data (Q, D are set, done by discovery thread). This is a complex operation. There can be multiple `peer` that have a `peer_ni` mentioned in the data.
    1. if more than one `peer` has D set
        1. retain only the newest set of data, as determined by the configuration sequence number
        2. clear D on every `peer` mentioned
    2. if discovery is disabled or C is set on any `peer`
        1. clear D
        2. set P if the sent data matches the current `peer`
        3. otherwise complain
    3. if discovery is enabled and C is not set on all `peer`s
        1. set P
        2. clear C if any changes are made to the `peer`
        3. create a new detached `peer_ni` if necessary
        4. detach any required `peer_ni` from an undiscovered `peer` (P not set)
        5. attach the detached `peer_ni`
        6. clear do-not-use state of these `peer_ni`
        7. copy L, R, A, state from the undiscovered `peer`
        8. a `peer_ni` attached to a discovered `peer` (P set) indicates a problem
            1. we can detach this `peer_ni` and attach it to the `peer` being discovered
            2. we can leave it attached to its current `peer`
            3. we can force rediscovery for the discovered `peer` by clearing it's P state
            4. complain about suspicious goings-on
7. The discovery thread does something like the following:
    1. get queued `peer` (Q is set)
    2. if A or R is set
        1. continue to the next `peer`
    3. else if S is set
        1. resize the MD buffers
        2. clear S
    4. else if D if set
        1. merge the received data
    5. else if discovery is disabled
        1. clear Q

        2. wake any waiters on the `peer`
    6. else if P is not set
        1. send a ping message
        2. the `peer` will be queued for work again once the ping reply comes in
    7. else if M is not set (not multi-rail capable)
        1. clear Q
        2. wake any waiters on the `peer`
    8. else if L is not set
        1. send a push message
        2. the `peer` will be queued for work again once the push ack comes in
    9. else
        1. clear Q
        2. wake any waiters on the `peer`

A couple of notes:

- The lifecycle of `peer_ni` and `peer_net` structures is strongly intertwined with that of `peer` structures.
- LNet currently creates a `peer` whenever it receives a message from an unknown sender, and those peers stick around indefinitely. Which raises the question whether DLC can ever truly delete a `peer`. For this design, the alternative would be that a `peer_ni` can be detached from a multi-rail peer via DLC, but that the `peer_ni` sticks around with its own `peer->peer_net->peer_ni` chain afterwards. (Or in a design that allows for "bare" `peer_ni` structures, it could stay in that form.)
- If discovery is enabled, and both Active and Passive are uprev, and the passive doesn't see a push message before it sends a message to the active, it ends up pinging the active node. If the active node sees that ping before it has sent the push message, the push message will not be sent at all.
- A peer created to receive a message from a downrev node will be discovered when the uprev node sends a message.
- A user-initiated ping also needs to queue a (6) Merge of the data received into the peer structures.
- Adding or deleting a local NI forces pushes to be sent to the peers. We may be able to do this lazily, but that only holds if at least one local NID is guaranteed to be stable. (This could be an identity-containing loopback NID, or it could be that there is a "primary" NID for which the NI cannot be removed.)
- Switching from discovery disabled to enabled triggers discovery of a peer the next time a message is sent to it.
- If we somehow fail to receive or store the data of a (5.c) Received push message, then we still clear P, which causes the node to ping the peer next time a message is sent.
- Whether a failure to send or receive a message during Discovery is recoverable depends on whether more than one `peer_ni` is already attached to the `peer` being discovered. If there is only one `peer_ni` then the entire `peer` is marked bad on failure. This generalizes to the case where there are multiple `peer_ni` for a `peer` but all of them turn out to be unhealthy.

The data that needs to be stored when receiving a ping reply or push message consists of at least the list of NIDs at 8 bytes per NID, so if we set LNET_MAX_INTERFACES at 16 that amounts to 128 bytes. If we want to be able to handle 128 interfaces this grows to 1024 bytes. The only warning we have that a push message might arrive is a previous ping, and this doesn't apply for later updates. Allocating such a buffer to each peer structure seems wasteful. The alternative is to allocate the holding buffer in the EQ callback. An EQ callback runs with the lnet_res_lock held, which is a spinlock. So GFP_ATOMIC must be used, and hence the allocation can fail. At that point we still have some useful information in the MD buffers, even if we cannot retain a copy, and can use this information to "tag" any peer_ni that are going away. This prevents us from trying to use any of those peer_ni NIDs to discover more for this peer. New peer

NIDs cannot be reliably added though, and I prefer to defer that work. We also know that the our peer config information is outdated.

The main reason for holding the lnet_res_lock seems to be that it prevents the EQ and MD from being deallocated from under us while the EQ callback runs. This implies that it *may* be safe to drop and re-take this lock in the callback, if we have ensured by other means that this cannot happen. I'd rather not have to do this though.

## Merging peer structures

Multiple peer structures `(peer, peer_net, peer_ni)` can be formed before discovery establishes that they all belong to the same peer. At that point these structures need to be merged. This is a quick sketch of what code combining the peer datastructures could look like – take it with a grain of salt, it is known to be incomplete:

```
typedef struct {
        lnet_nid_t ns_nid;
        __u32      ns_status;
        __u32      ns_unused;
} WIRE_ATTR lnet_ni_status_t;
typedef struct {
        __u32                   pi_magic;
        __u32                   pi_features;
        lnet_pid_t              pi_pid;
        __u32                   pi_nnis;
        lnet_ni_status_t        pi_ni[0];
} WIRE_ATTR lnet_ping_info_t;
/*
 * Process push data.
 */
int
lnet_process_peer_push_data(ln_ping_info_t *push_data)
{
    /*
     * if data for more than LNET_MAX_INTERFACES was sent, it will
     * have been truncated.
     */
    if (push_data->pi_nnis > LNET_MAX_INTERFACES /* value */) {
        complain();
        push_data->pi_nnis = LNET_MAX_INTERFACES;
    }
    /* Here we hold the mutex across the entire operation */
    lnet_mutex_lock();
    /* Check whether there is a peer struct already. */
    peer = NULL;
    lnet_net_lock(LNET_LOCK_EX);
    /*
     * lock not needed for lookup, except modifying the reference
     * counts, unless those are made atomic.
     */
    for (i = 0; i < push_data->pi_nnis; i++) {
        nid = push_data->pi_ni[i].ns_nid;
        peer_ni = lnet_peer_ni_lookup_locked(nid);
        if (peer_ni) {
            peer = peer_ni->pni_net->pn_peer;
            lnet_peer_addref(peer); /* ? */
            lnet_peer_ni_decref(peer_ni);
            break;
        }
    }
    lnet_net_unlock(LNET_LOCK_EX);
    if (!peer) {
        peer = lnet_peer_alloc();
        if (!peer) {
            rc = -ENOMEM;
            failure;
```

```
    }
}
/*
 * We have a peer struct. Walk the nid list again to create
 * the networks.
 */
for (i = 0; i < push_data->pi_nnis; i++) {
    nid = push_data->pi_ni[i].ns_nid;
    netid = LNET_NIDNET(nid);
    peer_net = NULL;
list_for_each(&peer->lp_nets, e) {
        pn = list_entry(e, pn_list);
        if (pn->pn_netid == netid) {
            peer_net = pn;
            break;
        }
    }
    if (peer_net)
        continue;
    /* Add new network. */
    peer_net = lnet_peer_net_alloc();
    if (!peer_net) {
        rc = -ENOMEM;
        failed;
    }
    lnet_net_lock(LNET_LOCK_EX);
    list_add_tail(&peer->lp_nets, &peer_net->pn_list);
    lnet_peer_addref_locked(peer);
    lnet_net_unlock(LNET_LOCK_EX);
}
/*
 * Now the peer has the networks. Check each ni.
 */
pni = NULL;
for (i = 0; i < push_data->pi_nnis; i++) {
    nid = push_data->pi_ni[i].ns_nid;
    netid = LNET_NIDNET(nid);
    cpt = peer_nid_to_cpt(nid);
    /* Pre-emptive allocation. */
    if (!pni) {
        if (!list_empty(ptable->pt_parked_peer_ni)) {
            pni = list_entry(ptable->pt_parked_peer_ni);
            list_del(&pni->pni_hashlist);
            memset(pni, 0, sizeof(*pni));
        } else {
            pni = lnet_peer_ni_alloc();
            if (!pni) {
                rc = -ENOMEM;
                failure;
            }
        }
    }
    /* Find the net. */
    peer_net = NULL;
    list_for_each(&peer->lp_nets, e) {
        pn = list_entry(e, pn_list);
        if (pn->pn_netid == netid) {
            peer_net = pn;
            break;
        }
    }
    LASSERT(peer_net);
    lnet_net_lock(LNET_LOCK_EX);
    peer_ni = lnet_peer_ni_lookup_locked(nid);
    if (peer_ni && peer_ni->pni_net != peer_net) {
        /* Attached to different peer struct. */
        pn = peer_ni->pni_net;
        list_move_tail(&peer_ni->pni_list, &peer_nets->pn_nets);
        peer_ni->pni_net = peer_net;
        lnet_peer_net_decref_locked(pn);
        lnet_peer_net_addref_locked(peer_net);
        if (list_empty(pn->pn_nis)) {
```

```
                p = pn->pn_peer;
                list_del(pn->pn_list);
                lnet_peer_decref_locked(p);
                list_move(&pn->pn_list, &ptable->pt_parked_peer_net);
                if (list_empty(p->lp_nets)) {
                    // get rid of peer...
                    list_move(&p->lp_list, &ptable->pt_parked_peer);
                }
            }
        } else if (!peer_ni) {
            /* No peer_ni, insert our new one. */
            peer_ni = pni;
            pni = NULL;
            /* Attach to peer_net */
            list_add_tail(&peer_ni->pni_list, peer_net->pn_nis);
            peer_ni->pni_peer_net = peer_net;
            lnet_peer_net_addref(peer_net);
        }
        lnet_net_unlock(LNET_LOCK_EX);
    }
    lnet_mutex_unlock();
    return rc;
}
```

# Finite State Machines

## Local NI FSM

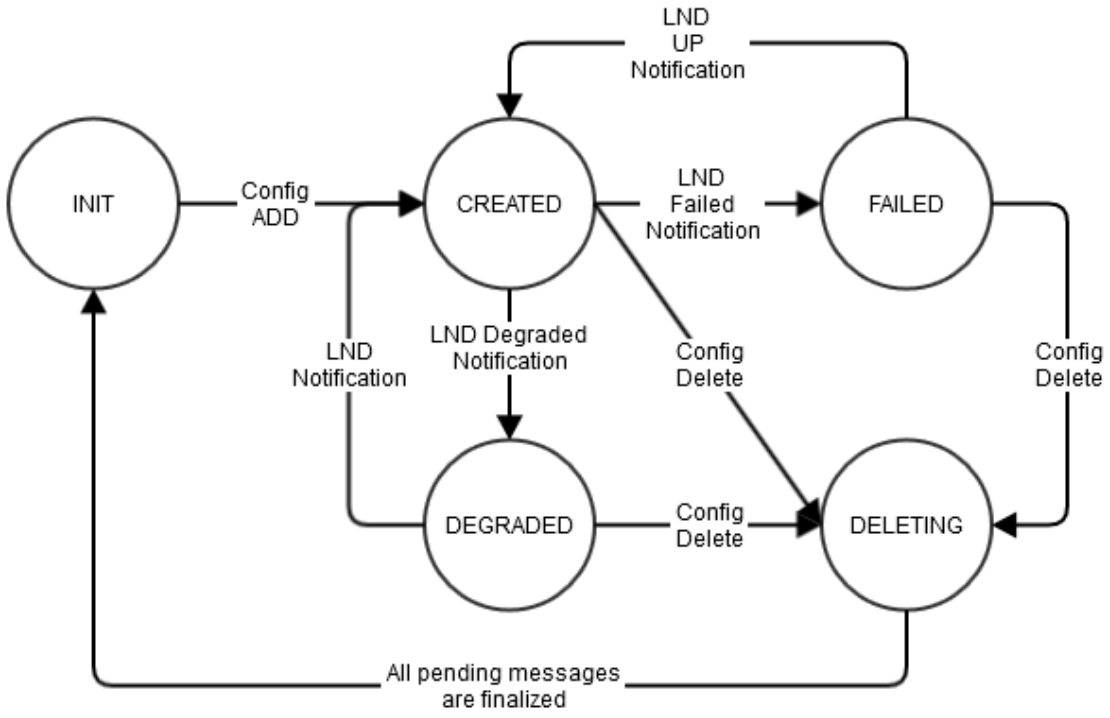



**Figure 10: Local NI FSM**

- INIT: pre state. Only transitory.
- CREATED: The NI has been added successfully
- FAILED: LND notification that the physical interface is down. hlt-005, hlt-010,
- DEGRADED: LND notification that the physical interface is degraded. IB and ETH will probably not send such a notification. hlt-015, hlt-020, hlt-025
- DELETING: a config delete is received. All pending messages must be depleted.

Both Degraded and Failed both need the LND to notify LNet. For degraded the LND could possibly query the type of the card and figure out the theoretical speed, then if the measured speed is below, then we can mark as degraded. snd-080

snd-085 - TODO: need to identify in the design how we deal with local NI failures.

## Local Net FSM

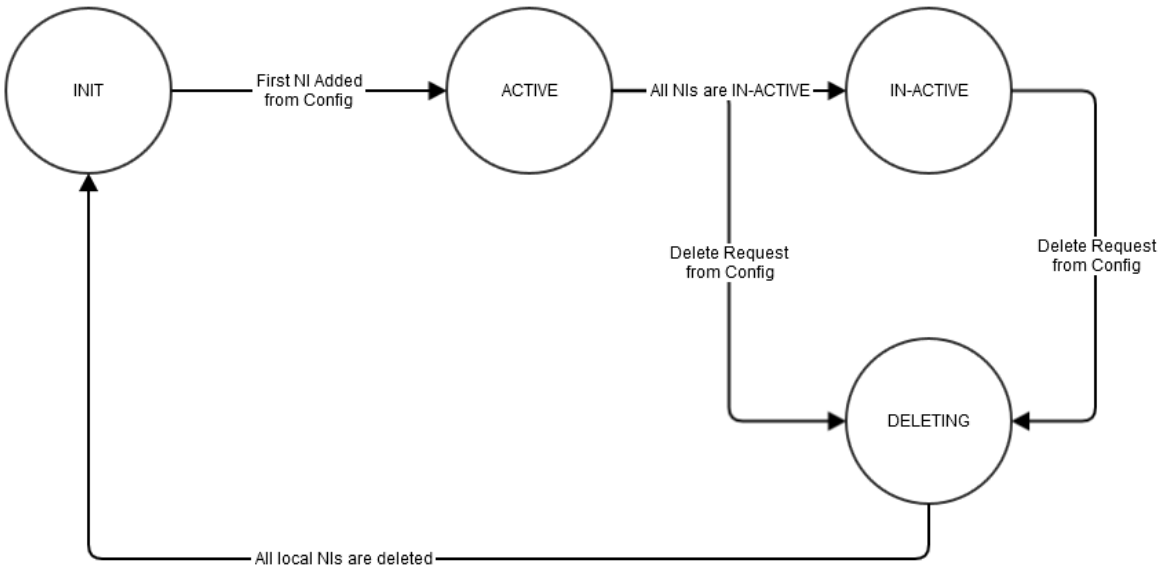



**Figure 11: Local Net FSM**

- INIT: pre state. Only transitory.
- ACTIVE: First NI has been added successfully
- IN-ACTIVE: All NIs are down via LND notifications.
- DELETING - Config request to delete local net

This FSM is driven from the Local NI FSM, since the Local Nets are implicitly created, by configuring the local NI.

## Peer FSM

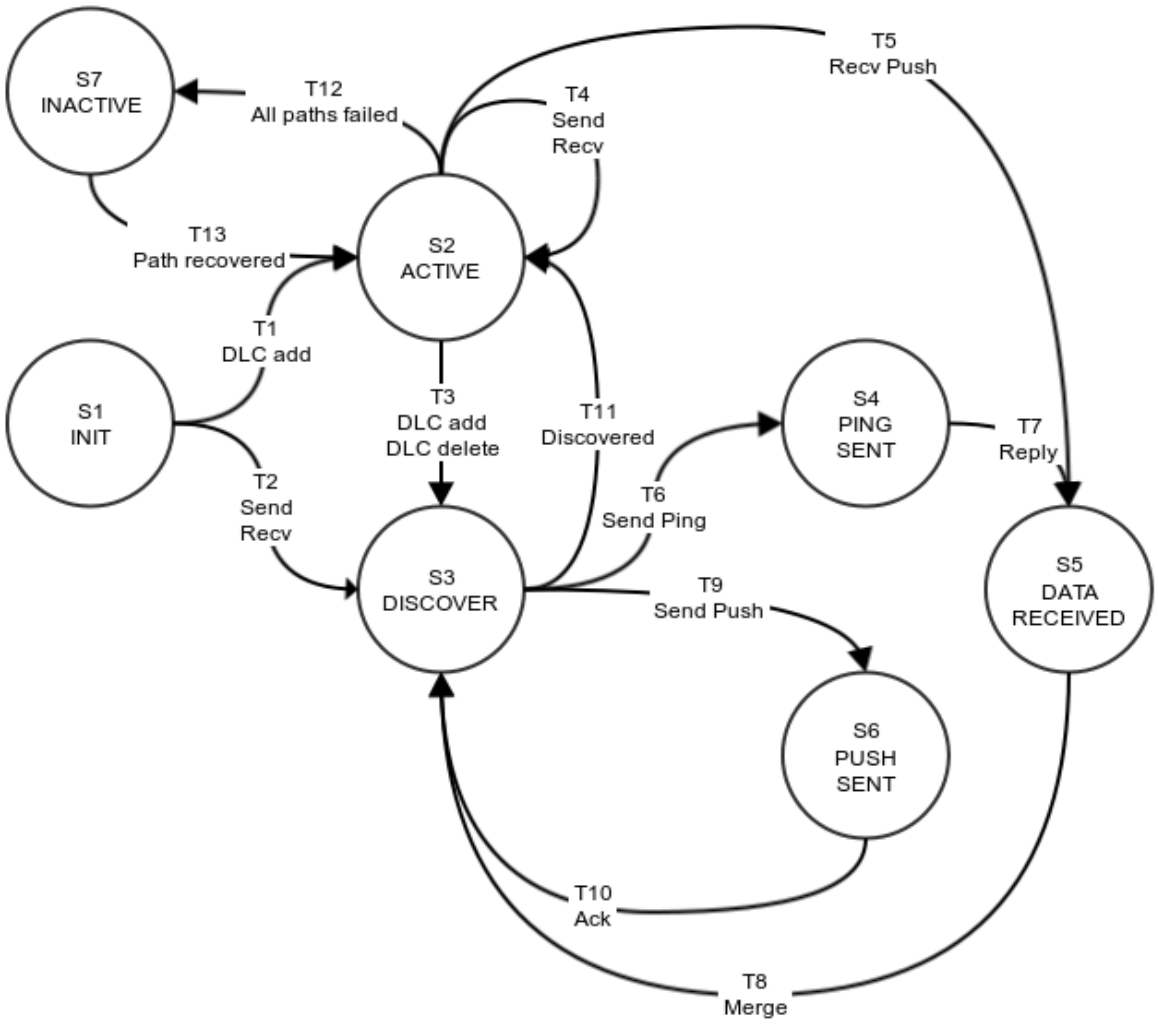



**Figure 12: Peer FSM**

This is a simplified FSM for a `peer` that illustrate how the various states relate to each other. The letters refer to the The Discovery Algorithm.

States

- S1: **INIT** initial state.
- S2: **ACTIVE** normal state for sending or receiving messages, discovery is either not needed or disabled.
  Q, D, A R clear. P, L are set if discovery is enabled, otherwise they may be clear.
- S3: **DISCOVER** being worked on by the discovery thread.
  P and/or L clear. D, A, R clear. Q set.
- S4: **PING SENT** a ping message has been sent to a peer, waiting for the reply
  Q, R set. P, A, D clear.

- S5: **DATA RECEIVED** ping or push data has been received
  Q, D set. P, R, A clear
- S6: **PUSH SENT** a push messages has been sent to a peer, waiting for the ack
  Q, A set. L clear.
- S7: **INACTIVE** all peer NI are marked bad, waiting to recover

Transitions

- T1: **DLC add** creating a `peer` through DLC
- T2: **Send / Recv** creating a `peer` from `lnet_send()` or `lnet_parse()`
- T3: **DLC add / DLC delete** modifying an existing `peer` by adding or deleting `peer_ni` using DLC, but also an existing `peer` going to discovery because earlier modifications were done with discovery disabled, and now discovery has been enabled
- T4: **Send / Recv** normal message traffic
- T5: **Recv Push** a Push message is received
- T6: **Send Ping** a Ping message is sent
- T7: **Reply** a Ping reply is received
- T8: **Merge** merge of data received from a Ping reply or Push message
- T9: **Send Push** a Push message is sent
- T10: **Ack** a Push ack is received
- T11: **Discovered** discovery is complete
- T12: **All paths failed** all paths to the `peer` are marked bad
- T13: **Path recovered** at least one path the `peer` is marked good again

An FSM reflecting the discovery algorithm in all its details would be too unwieldy to be of much use. Above is an attempt at a simplified FSM. TODO: explain how the diagram maps to the algorithm described above.

## Peer NI FSM

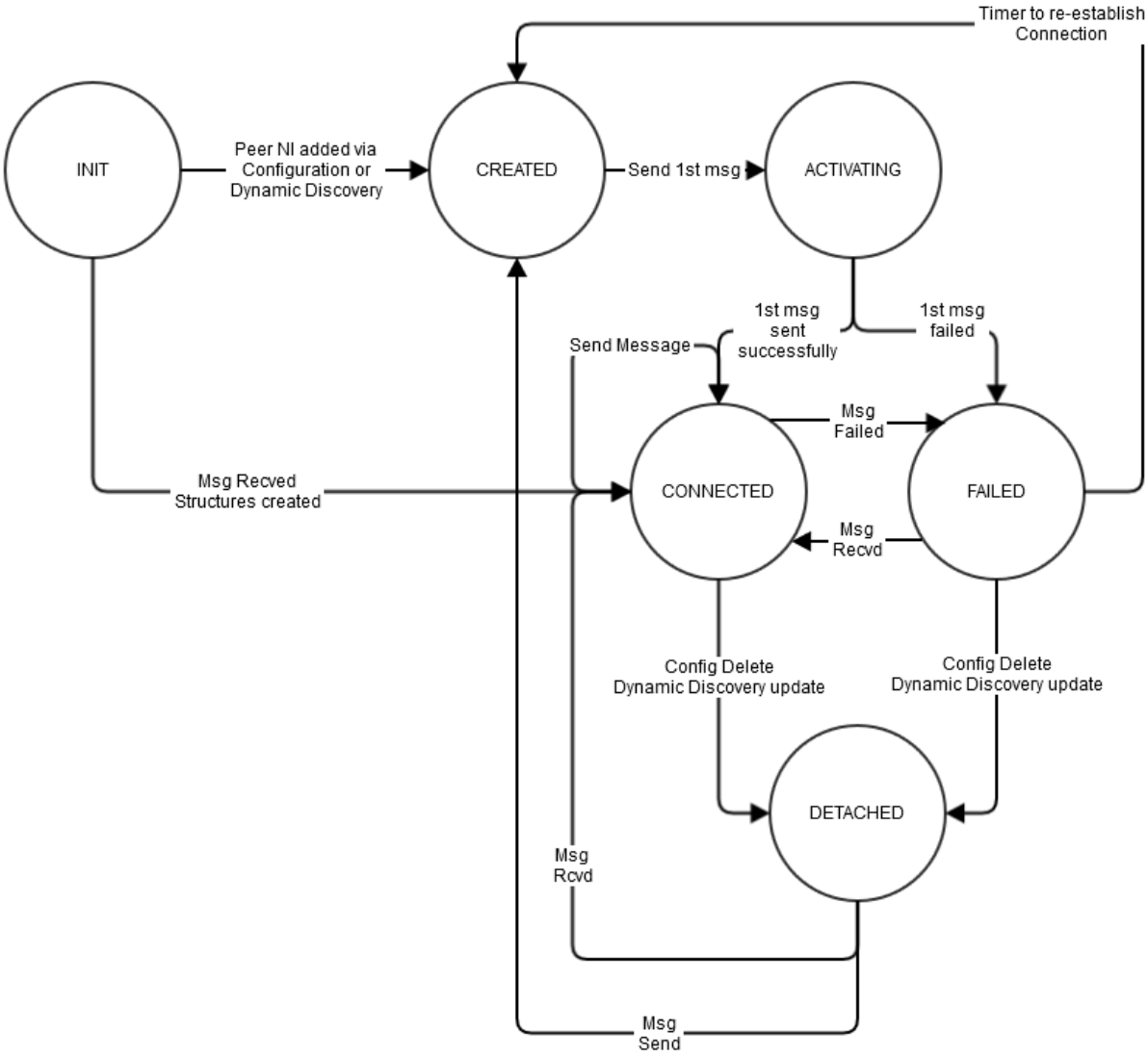



**Figure 13: Peer NI FSM**

When a peer_ni is initially added to the peer, it will not be in CREATED state, which means there is no active connection with that peer_ni.

When that peer_ni is selected for first usage it will go into ACTIVATING state. If the message is sent successfuly, then it'll move to CONNECTED state, otherwise it will move to FAILED state.

Subsequent messages will be happen in the CONNECTED state. If any message fails from then on, it moves to the FAILED state.

When a connection failed the peer_ni is put on a list which is checked periodically by another thread which reinitiates the connection.

- INIT - pre state. Only transitory
- CREATED - peer_ni created but no active connections exists.
- ACTIVATING - 1st message sent to the peer_ni, but has not completed yet
- CONNECTED - 1st message sent successfully
- FAILED - A message (1st or after) has failed to send
- DELETING - A dynamic update or a config delete removes that peer_ni