Lustre Networking over MPI

BY ERIC BARTON

AN ORNL LUSTRE CENTRE OF EXCELLENCE WHITE PAPER
AUGUST 2007
VERSION 1

Abstract

The network subsystem of the Lustre® file system provides an abstract communications API, layered over network-specific drivers. This paper describes the design of a Lustre network driver (LND) for the Message Passing Interface (MPI). It is intended to provide Lustre support on HPC clusters where a native LND is not available.

Contents

1. Introduction

Lustre networking (LNET™) uses a layered architecture to provide an abstract communications API, independent of the different networks on which it runs. Each supported network type has its own Lustre Network Driver (LND) which presents a standard internal interface to the upper layers of LNET and uses the native network interface below it.

New HPC cluster networks with their own non-standard interfaces require their own LND before Lustre/LNET can run "natively" on them. Since the Message Passing Interface (MPI) is usually one of the first standard communications APIs to be supported on new HPC cluster networks, layering LNET over MPI will provide Lustre support if a "native" LND is unavailable.

1.1 Requirements

The MPI LND must satisfy the following requirements:

- Support Lustre for HPC applications
- Maximize portability
- Provide predictable fault handling
- Enable transparent startup/shutdown
- Allow no interference with MPI_COMM_WORLD
- Support routed LNET networks
- Provide static server/router network addressing
- Support both single-threaded and multi (POSIX)-threaded client runtimes

Lustre networking over MPI is both a porting strategy and a risk mitigation strategy since it provides a stopgap until native LNDs have been developed and debugged. It therefore relies wherever possible only on the most common MPI features to maximize portability and to minimize the likelihood of exercising bugs in MPI implementations it uses. Overall, the implementation should prefer portability and robustness over performance. It should therefore restrict itself to MPI-1 API calls unless specific MPI-2 features are an absolute requirement. The implementation should also avoid using MPI in "unfamiliar" ways. For example, zero-copy bulk transfers are possible with MPI. However, a copy may be required on the server-side with some MPI implementations since mapping assumptions on RDMA-capable fabrics may be broken if MPI is passed kernel buffers that have been mapped into userspace.

Historically, MPI has been used for parallel applications where all processes start and stop together, and communication errors are handled simply by aborting the whole application. Although newer MPI implementations may support better error handling or even full fault tolerance, reliance on these features contradicts the portability requirements described above. Therefore, the only requirement for fault handling is that it should be predictable - e.g. it should either abort the whole application or return an error to the application.

MPI requires communicating processes to share a common communicator. The default communicator, MPI_COMM_WORLD, is used by parallel application processes to enumerate and locate their peers. Attempts to "subvert" user-level communicators (e.g. by replacing MPI_COMM_WORLD with a communicator that only spans the application processes) must be avoided for portability. Therefore, the MPI-2 accept/connect APIs are required to establish connectivity to the server-side processes.

Lustre server and LNET router network addresses are bound to physical nodes. These addresses do not change between system reboots or depend on which nodes are currently in service. MPI addresses processes by rank within a communicator – there is no fixed mapping to hardware. The implementation must, therefore, provide a lookup table for clients to convert absolute server-side addresses to a corresponding MPI inter-communicator rank. The implementation may also perform network address translation on client LNET addresses to allow Lustre servers and LNET routers to distinguish concurrent applications. However, it may not translate the NET field of these addresses so that LNET route table administration on connected non-MPI networks remains simple.

The MPI LND should target both single-threaded and multi-threaded runtimes. In a multi-threaded runtime, a background thread to handle network events will ensure client processes remain responsive to network events at all times – even when the application is "crunching". However this depends also on the thread-safety of the MPI implementation being used.

1.2 Architecture

LNET communications over MPI are supported by two new network drivers, the MPI LND and the userspace proxy (UP) LND. The MPI LND is a standard userspace LND that allows a liblustre application to layer LNET protocols over MPI. The UP LND is a "glue" driver that plugs one or more instances of a userspace LND into kernel LNET.

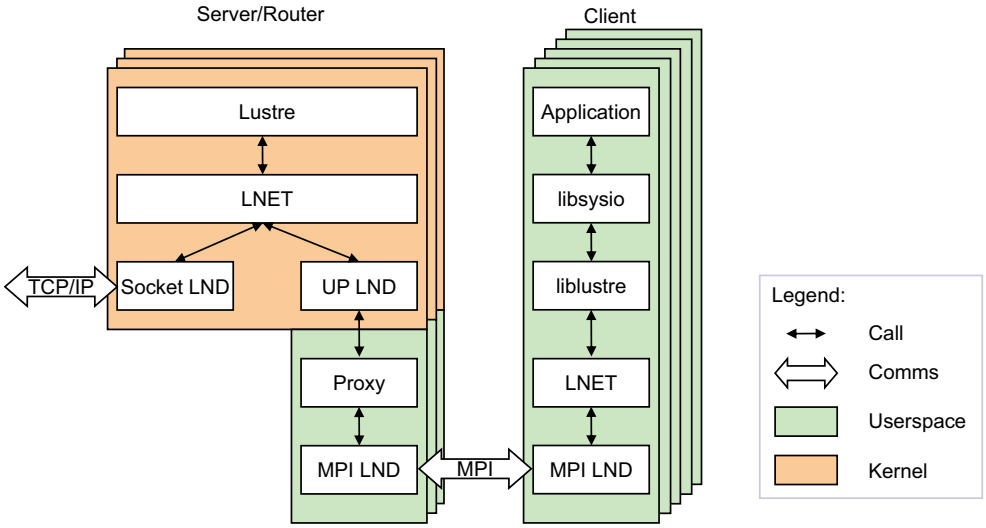
1.2.1 Process Structure

Communications between a parallel application using the MPI LND and kernel LNET are achieved by instantiating the UP LND on all the Lustre servers and LNET routers with direct connectivity to the application. This set of processes forwards LNET communications between the kernel and userspace LNETs. It is called a "parallel proxy". Figure 1 illustrates the structure.

By default, MPI communications failures are fatal for both the client application and the parallel proxy. All fault handling is, therefore, done in the UP LND. When it detects that a connected userspace process has died, it terminates all outstanding communications with that process and releases all resources allocated to it.

Each client application should instantiate its own parallel proxy to provide fault isolation. The mechanisms to achieve this are probably specific to different MPI run-time systems. Some alternate implementations are suggested in Section 2. Implementation.

Figure 1. Parallel proxy structure



1.2.2 Network Addressing

The LNET MPI network uses a static NID address space for proxy processes based on their primary IP address so that individual proxies may be failed out.[1] These processes use the standard Lustre/LNET server PID.

The LNET MPI network uses a dynamic NID address space for client processes based on their MPI rank since these processes are not persistent. All the client processes of a single application are assigned the same LNET client PID which the UP LND uses to distinguish concurrent applications.

When a client application establishes connectivity with its parallel proxy, the 0^{th} proxy process requests a new PID from its UP LND and broadcasts this to its parallel proxy peers, which then register the PID with their own UP LNDs. At this time, all proxy processes query their own NID from the UP LND to create a local (rank, NID) tuple. All these tuples are gathered on the 0^{th} process and then broadcast to the client application. The client application uses this to construct a routing table that converts proxy NID to parallel proxy process rank.

2. Implementation

2.1 Establishing Communications

Communications between the client application and the parallel proxy occur via an MPI inter-communicator. This is set up in the client from __liblustre_setup_, which in turn initialises LNET including the MPI LND. The application programmer must treat this as a collective operation on MPI_COMM_WORLD and must call it after MPI_INIT[2].

Creating a proxy and establishing connectivity with the proxy is strongly dependent on the MPI implementation. Two major alternative implementations, with variations on the first, are described below. Unfortunately, they all rely to some extent on MPI-2 features and each has its own problems.

1 A parallel proxy server is instantiated at system startup. It uses the MPI-2 client/server procedures to establish the shared inter-communicator. The server opens an MPI port and waits to accept connections on it. The port name is passed to the client application using name publishing if the MPI implementation supports it. Otherwise, a site-specific mechanism (e.g., via the process environment) is required.

   After the client has been accepted, there are further choices.

   a. The server forks a copy of itself everywhere. The child becomes a dedicated parallel proxy for the client application and the parent continues to accept new connections on behalf of other applications.

      Although this is the most "natural" implementation, it depends on the child processes being able to inherit and use the MPI state from the parent – a feature which many MPI implementations do not support.

   b. The server uses MPI_COMM_SPAWN to instantiate the parallel proxy. On startup, the newly-spawned parallel proxy opens a new MPI port, communicates its name to its parent, and the parent communicates this port name back to the client application. The client application then closes its connection with the original server and establishes a connection with the newly spawned parallel proxy.

      This implementation depends on the server being able to use MPI_COMM_SPAWN to instantiate the parallel proxy on all the required nodes.

1. Note that such failures are fatal for any parallel applications with parallel proxies running on these nodes at the time of the failure.
2. At startup, liblustre performs a collective initialisation over all client processes on the assumption that any and all client processes may conduct Lustre I/O. An initialisation API taking an explicit communicator could be provided if the application will only do Lustre I/O from a subset of its processes.

c. The server does not try to create a copy of itself. The server is the parallel proxy and it serves all its clients concurrently.

   This implementation does not achieve fault isolation. An error in one application may cause all currently running applications to abort.

2 Each application instantiates its own parallel proxy using MPI_COMM_SPAWN. This requires the MPI implementation to interpret information passed to MPI_COMM_SPAWN via the "info" parameter so that the processes are spawned on the correct Lustre server/router nodes, not the "normal" compute nodes.

2.2 LNET Changes

The LNET/LND interface may be changed to allow the UP LND / MPI LND to cooperate better on receiving "immediate" messages (i.e., messages that include the full payload) as follows:

- Add immediate payload buffer parameters to lnet_parse. These should be 0/NULL with non-immediate payloads.
- Any LND passing an immediate payload must also implement a new callback, lnd_delayed_immediate_done.
- When an immediate payload is passed to lnet_parse, the return codes are < 0 on error, 0 on a successful immediate match and EAGAIN if the receive must block for a matching receive buffer (e.g. when router buffers are all busy or the receiving portal is lazy and there are no matching MEs).
- If lnet_parse with an immediate buffer must block the receive, lnd_eager_recv (if defined) is called as usual. lnd_delayed_immediate_done is called when the receive finally completes.

This change will only be implemented permanently if justified by performance measurements.

2.3 MPI LND

The MPI LND uses point-to-point, two-sided, unbuffered, non-blocking MPI communications exclusively. There are two message classes depending on message length. Short message handling tries to minimize latency while large message handling tries to maximize bandwidth.

Short messages are sent unsolicited. Each MPI LND instance has a pool of short message buffers, posted using persistent communication requests with a reserved tag to receive these messages. Payload is copied at the receiver on arrival and the receive buffer is reposted. The pool is sized just large enough to maximize throughput. The MPI LND relies on MPI flow control to handle contention fairly when the pool is temporarily exhausted.

Large messages are sent in two pieces. The payload is posted for sending using a unique tag, and the LNET header plus unique tag is sent using the short message pool described above. When the short message is received at the destination, the target buffer is identified using the LNET header and it is posted for receive using the unique tag. This allows MPI to transfer the payload and signal completion to both sides.

Optimized LNET GETs (which elide the REPLY message) are handled very similarly. However, since the payload transfer is towards the sender, the sink buffer can be posted with a unique tag before the GET request is sent. When the GET request is received and the source buffer has been found, it is sent using the unique tag included with the GET request. In this case, the payload may be sent with a "ready" send, which some MPI implementations may be able to optimise over a "normal" unbuffered send.

The first message an MPI LND instance sends to any peer is a version query/response containing version and usage information. The receiver will reply with a similar message. No other type of message may be sent to a peer before a response has been received.

The server-side must pass message payload into the kernel. This unfortunately requires a copy since mapping kernel buffers into the user address space would most probably break MPI implementations on RDMA-capable networks[3].

The MPI LND implements LNetEQPoll with a zero timeout using MPI_TESTSOME. With a non-zero timeout, it uses MPI_WAITSOME. This may block indefinitely, effectively ignoring Lustre timeouts.

The server-side MPI LND should try to ensure MPI_WAITSOME does not busy-poll since it is not the only consumer of CPU resource on the node. The implementation of this depends on the particular version of MPI being used.

The MPI LND expects MPI to abort on error. However it checks the status on return from all MPI calls it makes in case MPI has been configured to return errors rather than abort. It has its own tuneable to determine whether to complete relevant communications with error or to abort the whole application if MPI returns an error.

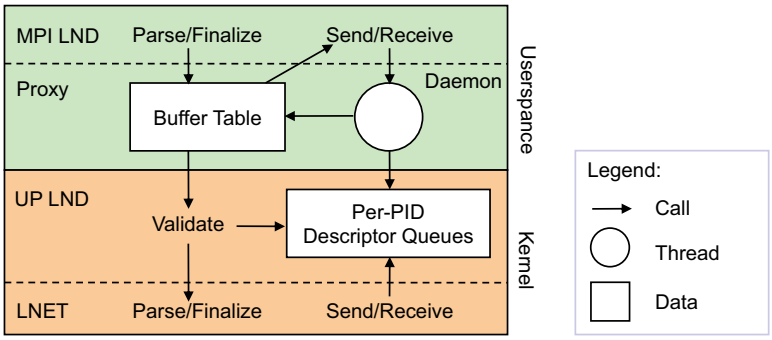
2.3.1 Parallel Proxy

The UP LND appears to LNET as a standard LND. It registers itself when the module loads and on LNET startup, its own startup routine is called to tell it which LNET network(s) it must proxy for and to set its own NID from the node's primary IP address.

Proxies communicate with the UP LND via ioctl() on /dev/lnet as shown in Figure 2. When a new parallel proxy starts up, it opens the ioctl device, only closing it on shutdown. This allows the UP LND to clean up any uncompleted communications should the parallel proxy terminate abnormally.

The parallel proxy then requests or registers the PID for its application processes (note that this is the PID described in Section 1.2.2 Network Addressing, not the local process PID) and gets its NID, which it combines with the NIDs of its peers to create the routing table of the client-side MPI LND. It also creates pools of large and small send buffers[4]. The send buffer descriptors are posted to the UP LND ready for outgoing sends from kernel LNET.

Figure 2. Proxy communication with the UP LND



The per-PID state in the UP LND includes queues of pending and active send and receive descriptors. Pending descriptors are waiting for related communications to be initiated in userspace, while active descriptors are waiting for related communications to complete in userspace. Pending descriptors are moved to the active list and handled in userspace on return from all UP LND ioctls. The parallel proxy dedicates one or more daemons to this to ensure the pending queues are consumed eagerly. The daemons simply block interrupt-ability in the kernel while the pending queues are empty.

3. An MPI implementation for an RDMA-capable fabric has to keep the application's virtual memory map in synch with the network device. This can be done by mapping all memory at startup and then keeping it in synch by intercepting system calls that modify the address space – typically on memory allocation. It is very unlikely that MPI implementations could keep the network device in synch if arbitrary kernel buffers are mapped into the address space when the application calls into the UP LND.
4. If the performance of large "immediate" receive buffers in the server-side MPI LND does not justify the LNET changes required to support them, a pool of large and small receive buffers will also be allocated at this time.

Send and receive paths are handled as follows:

- **Immediate Receive** (assuming LNET changes to support immediate receive are implemented)
    1. When the MPI LND calls lnet_parse, passing the immediate buffer containing the message payload, a userspace receive descriptor is allocated and the call is forwarded to the UP LND. The UP LND allocates a kernel receive descriptor, adds it to the active receive list, and calls lnet_parse with a NULL immediate buffer pointer, the kernel descriptor handle, and the LNET message header.
    2. If kernel LNET can receive immediately, it calls lnd_recv in the UP LND, which copies in the payload data from userspace and calls lnet_finalize to signal completion. When lnet_parse returns signalling completion, the kernel receive descriptor is freed. Control is returned to userspace where the userspace receive descriptor is freed, and lnet_parse returns to the MPI LND.
    3. If kernel LNET must delay the receive, lnet_parse returns EAGAIN. Control returns to userspace and lnet_parse returns EAGAIN to the MPI LND.
    4. When kernel LNET finally matches the receive, it calls lnd_recv in the UP LND, which copies the payload data from userspace, calls lnet_finalize to signal completion, and moves the kernel receive descriptor to the pending receive queue.
    5. When the receive descriptor is ready to be handled (a returning UP LND ioctl removed it from the head of the pending receive queue), it is added to the active receive list. On return to userspace, the corresponding userspace descriptor is located and the MPI LND's lnd_late_immediate_done callback is called to signal completion.
- **Receive** (if LNET changes to support immediate receive are not implemented)
    1. When the MPI LND calls lnet_parse to receive an incoming message, a userspace receive descriptor is allocated and the call is forwarded to the UP LND, which in turn calls lnet_parse, passing the descriptor handle and the LNET message header.
    2. When kernel LNET calls the UP LND to receive the payload, a receive descriptor is allocated and queued on the pending receive queue.
    3. When the receive descriptor is ready to be handled (a returning UP LND ioctl removed it from the head of the pending receive queue), it is added to the active receive list. On return to userspace, the corresponding userspace descriptor is located and a receive buffer is allocated. If all receive buffers are busy, the userspace descriptor is queued until a receive buffer is freed. Otherwise the MPI LND is called to receive the payload.
    4. When the MPI LND calls lnet_finalize to complete the receive, it calls into the UP LND which locates the original receive descriptor on the active receive list. The UP LND copies the payload from the userspace buffer into the kernel buffer described by the kernel receive descriptor, calls lnet_finalize to signal completion, and then frees the kernel receive descriptor. On return to userspace, the userspace receive descriptor is freed, and the receive buffer is returned to the relevant pool.
- **Send**
    1. When kernel LNET calls lnd_send in the UP LND, a send descriptor is allocated. If all send buffers are busy, the send descriptor is queued until one becomes free. Otherwise a buffer is allocated, the payload is copied out to userspace, and the send descriptor is queued on the pending send queue.
    2. When the send descriptor is ready to be handled (a returning UP LND ioctl removed it from the head of the pending send queue), it is added to the active send list. On return to userspace, the MPI LND is called to send the message.
    3. When the MPI LND calls lnet_finalize to complete the send, it calls into the UP LND which locates the original send descriptor on the active send list. The UP LND marks the send buffer free, calls lnet_finalize to signal completion, and frees the kernel send descriptor.

3. Development and Test Plan

The initial version of the MPI LND will implement all communications over MPI, but not connection establishment to the parallel proxy. In this version, the IP-to-inter-communicator rank route table will be empty and all MPI NIDs will be interpreted as peer clients. Initial testing will use a throw-away userspace test application that initialises MPI and LNET, executes an MPI barrier, and then pings a peer. Large message payloads will not be tested at this time. This development can proceed independently of server-side issues.

Parallel proxy instantiation and connection establishment will be explored initially using purpose-built MPI test programs to determine the best implementation. Note that alternative implementations may be required for different MPI runtimes.

Initial testing of the full system, including connection establishment with the parallel proxy, will use a small throw-away test application. This will initialise MPI and LNET, execute an MPI barrier, and then ping the parallel proxy nodes from the application nodes.

The next stage of testing will use LNET self-test, including the standard userspace LNET self-test client linked with the MPI LND. This will test all types of LNET communications and provide the first performance statistics. These tests must include:

- Routed configurations
- Powering off/rebooting any parallel proxy or application node
- Aborting/crashing application processes
- Killing parallel proxy processes

The final stage of testing will run the Lustre test suite.

Legal Disclaimer

Lustre is a registered trademark of Cluster File Systems, Inc., and LNET is a trademark of Cluster File Systems, Inc. Other product names are the trademarks of their owners. Although CFS strives for accuracy, we reserve the right to change, postpone, or eliminate features at our sole discretion.