

Portals and Networking for the Lustre File System

Peter J. Braam, Phil Schwan
Cluster File Systems, Inc.
530 Showers Dr # 7-147
Mountain View, CA 94040
{braam,phil}@clusterfs.com

Ron Brightwell*
Sandia National Laboratories
PO Box 5800
Albuquerque, NM 87185-1110
bright@cs.sandia.gov

Abstract

Lustre is a scalable parallel file system for use on large-scale compute clusters. Lustre needs to support many different networks, including Ethernet, InfiniBand, Myrinet and Quadrics. In this paper, we discuss how the Portals message passing API developed by Sandia National Laboratories provides a nearly optimal foundation for networking the Lustre file system. Of particular interest are the concurrent presence of storage networking features, such as remote DMA and request processing features that Portals allows.

1. Introduction

The current Portals data movement interface [5] is an outcome of research into high-performance message passing in lightweight kernel operating systems for massively parallel distributed memory parallel computing platforms. Earlier generations of Portals were implemented in the Sandia/University of New Mexico Operating System (SUNMOS) [9] and its successor, the Puma [11] operating system. Both of these lightweight kernels were deployed on large-scale production platforms with thousands of processors. A productized version of Puma, called Cougar, is currently in use on the 9000+ processor Intel ASCI/Red machine.

Over the last few years we have transitioned Portals technology from proprietary vendor parallel platforms to large-scale commodity-based PC clusters such as Sandia's Computational Plant (Cplant™) [4]. We redesigned the interface and semantics to support scalable, high-performance data movement using intelligent network interfaces such as Myrinet [3]. An important characteristic of Portals is that

*Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

it provides low-level, flexible building blocks that be combined to support higher-level protocols efficiently. In particular, we believe Portals supports the functional requirements of high-performance file systems like Lustre very well.

The rest of this paper is organized as follows. Section 2 describes the internal structure of the Portals 3.0 implementation. Section 3 discusses the networking infrastructure of Lustre, and Section 4 describes several current implementations.

2. Portals Internals

2.1. Protection Domains

The Portals API is not only useful to user-level systems like MPI or file system libraries, but also to kernel-level subsystems: this environment defines the protection domain of the *API*. The API will universally need *library (lib)* support for particular networking subsystems and this support will normally run in a different protection domain. Two important library protection domains are a kernel-level library that interacts with a device driver or networking library and a library resident on a network interface card.

2.2. Network Abstraction Layer (NAL)

NALs exist for both the API and lib domains. These abstraction layers are two virtual classes, one for use by the API, one for use by the lib domain. Derived classes are implemented by drivers usually associated with particular network systems, and instances of these classes are associated with a network interface offered by a particular network transport. Figure 1 illustrates the structure of the NAL implementation.

An instance of the Portals service therefore breaks down into the following components:

1. Portals *API library*: offers the Portals API to applications and interacts with the API NAL.

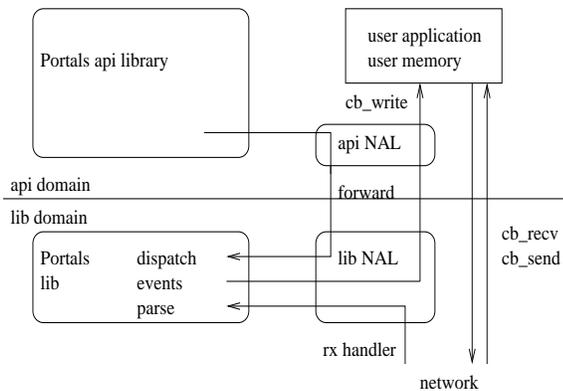


Figure 1. Portals

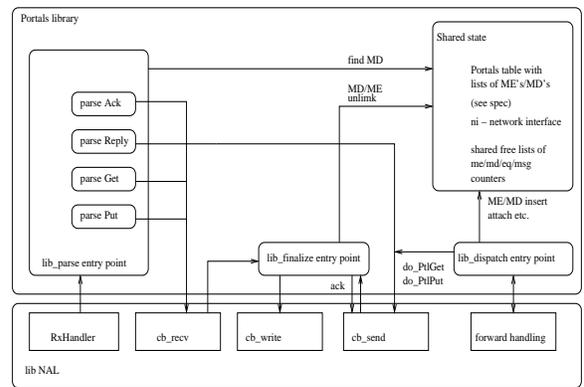


Figure 2. Portals Library Interactions

2. The **Portals library**: the library offers a method table including the following fundamental functions: `lib_dispatch()` executes functions forwarded from the **API library**, `lib_finalize()` is responsible for generating events visible to the **API library**. Finally `lib_dispatch()` is offered to find the purpose and destination of incoming packets.
3. The **API NAL** supplies a method table to the API library. One of the methods is the `forward` method. The **API library** uses the `forward` method as a function call interface to the **Portals library**.
4. The **lib NAL** is the final component. It dispatches requests forwarded by the **API NAL** to the **Portals library** and has several other functions. It has an autonomous handler associated with incoming traffic. It offers several methods to the **Portals library**: `cb_send()` is used to send packets, `cb_write()` crosses the protection domain and delivers both data and events to memory areas in use at in the API domain.

Figure 2 illustrates the internals of the Portals library and NAL library layers in more detail.

3. Lustre Networking Infrastructure

3.1. The File System

Lustre [1] is a clustered file system that combines features from many scalable previous distributed file systems with ideas derived from traditional shared storage cluster file systems. Lustre clients run the Lustre file system and interact with object storage targets (OSTs) for file data I/O and with metadata servers (MDS) for namespace operations. When client, OST and MDS systems are separate,

Lustre appears similar to a cluster file system with a file manager, but these subsystems can also all run on the same system, leading to a symmetric layout.

At the root of Lustre is the concept of object storage [6]. Objects can be thought of as inodes, and OSTs provide the file I/O service in a cluster. The name space is managed by metadata service which manages the inodes. Such inodes can be for directories, symbolic links or special devices, in which case the associated data and metadata are stored on metadata servers. When an inode represents a file, the metadata merely holds references to the file data objects stored on the OSTs.

Fundamental in this design is that the OSTs perform block allocation for data objects, leading to distributed and scalable allocation metadata. The OSTs also enforce security regarding client access to objects. The client OST protocol bears some similarity to systems like DAFS in that it combines request processing with remote DMA.

The client metadata protocols are transaction-based and are derived from AFS [7], Coda [8], and InterMezzo [2] file systems. The protocol features authenticated access, and write-behind caching for all metadata updates.

Lustre can provide UNIX semantics for file updates. Lock management supports coarse locks and for entire files and subtrees, when contention is low, as well as finer locks. Finer locks appear for extents in files and for pathname locks to enable scalable access to the root directory of the file system. All subsystems can transparently fail over to other nodes.

3.2. Naming

The first issue that needs to be addressed is the naming of systems, networks, and network interfaces and routing. Lustre uses a resource database that names filesets that are mounted, MDS servers that serve these, networks on which the service can be reached and descriptors of the network-

ing interface cards with the addresses under which they can be reached. As such, we implement a statically routed description of the cluster.

A client system offers its identity to the resource database and finds out what filesets to mount. Following the relational structures between the configuration descriptors, the client derives what network and network addresses to use. In the case a service migrates during a cluster transition, the client re-queries the resource database and adjusts its network destinations before transparently failing over to a replacement service. Services can be offered and used simultaneously on multiple networks.

The Portals API uses a triple (process id, network id, network interface) to identify a network destination. For kernel use, the process id is always 0, so an endpoint is identified by a network id and interface. It is network dependent if such an address can be reached through a connectionless protocol or if a connection needs to be established. A user-level connection agent on the client establishes connections to new systems and cleans them up in case of failures. After possible connection establishment, the kernel is handed the addressing information of the peer and proceeds to use the Portals addressing scheme to reach that system.

3.3. Request Processing

For the most part Lustre follows a remote procedure call (RPC) model of request processing. However, requests travel in both directions. For example, a client must handle lock callback requests if an OST or MDS server is making such requests, thereby acting as a RPC client.

The features of the Portals API are extremely well-suited to the request processing environment used by the Lustre subsystems. Incoming requests are managed by a ring of buffers. Portals provides for the automatic unlinking of such buffers when they are full. The request processing subsystem tracks when the incoming buffer is no longer in use by the request handlers and then re-enters the buffer in the ring. Associated with the incoming request buffer is an event queue in which events are delivered describing at what offset and from what originating system a request was delivered, and activating a handling thread if it isn't already working on requests. A service is also responsible for sending out reply packets. An event is generated in the sent reply event queue when a reply packet has been sent, so that allocated memory can be freed. On the client, when such a request is sent out, a reply buffer is preallocated and the server is told the match entry for the reply packet. There are event queues for sent requests and received replies.

When large numbers of related requests need to be made, for example for lock revocations, *multi-RPC's* are used. These send out multiple requests, without waiting for replies. When all requests are sent, replies are pro-

cessed. After a timeout, systems that have failed to reply are deemed to have left the cluster. This is in contrast with handling replies one by one, in which case multiple timeouts can be encountered and server parallelism is not exploited.

3.4. Bulk Transport

Some Lustre operations, such as reads and writes of object data on the OST, involve the transfer of large buffers during request processing. This bulk data movement is independent of the RPC and is described by two basic structures, a bulk descriptor and one or more network I/O buffers (niobufs). Both are symmetric structures used to enable data movement in either direction.

The node that will receive the data, the bulk sink, reserves the necessary memory area and builds a bulk descriptor. This descriptor contains all of the information that Portals needs in order to receive the data directly into that memory, as well as a mechanism for notifying the caller when the transfer completes. This mechanism relies on the Portals acknowledgment facility to receive notification upon successful message delivery, and is the only Lustre component to do so. The sending node, the bulk source, builds a nearly identical descriptor.

niobufs fall into two categories: local niobufs that describe an area of memory, and remote niobufs which describe a logical object extent. When Lustre needs to transfer multiple extents of a single object, it sends a vector of niobufs. Each niobuf represents a single non-overlapping extent.

In a typical object write, the source will create local niobufs pointing to kernel mapped pages, send remote niobufs with offset and length information, and commence bulk delivery. As the buffers are delivered, the source receives acknowledgements that trigger page unmapping and niobuf cleanup. The sink will receive the remote niobufs, map the relevant pages into memory, then wait for the data to arrive. As each buffer is received, a handler is notified and can perform any necessary finishing and cleanup.

Remote niobufs also contain space for additional delivery information, for example tokens to enable remote DMA. When a client wishes to engage in remote DMA transfers, it registers a memory buffer with the network interface and receives an opaque token which is added to the niobuf. The remote peer can use that token to directly access the memory on the client and perform faster I/O with lower overhead.

3.5. Polling and Event Handling

Contrary to most user-level message passing systems where polling the network is common, file services are typically event driven. The simplest events consist of waking up

a service thread or a thread waiting for a reply. Inline handling of small requests at event delivery time is becoming increasingly popular.

Portals events are delivered by the lib NAL, and this NAL may not be running in the protection domain in which the event can be executed. In fact, NALs implemented on intelligent network interface cards (NICs), described in more detail in the next section, can deliver events without generating any interrupts. Lustre has been careful to optionally accommodate polling clients by arranging event dispatch from a polling thread, but this is not an acceptable solution in most applications of the Lustre file system. We have made minor modifications to original Portals API and the NAL method table to optionally support generic inline callback processing when desirable.

4. Network Abstraction Layers

4.1. NIC-Based NAL's

Development of several NIC-based NAL's is currently in progress. Sandia has developed a Myrinet Control Program (MCP), which runs on the processor on the Myrinet NIC, that implements all of Portals' objects and semantics. This implementation is able to completely offload processing of messages from the host processor to the NIC and delivers all messages directly from the NIC to user-space buffers.

Sandia is also currently working on a NAL for the Quadrics [10] ELAN3 network using the Elan library interface. This NAL will use a queued DMA structure and a thread executing on the thread processor on the ELAN3 NIC to perform processing of Portal messages.

4.2. Library-Based NAL's

Library-based NALs, whether they reside in the kernel, on the NIC, or anywhere else, all export the same simple primitives. Fundamentally, the NAL is tasked with sending and receiving data on the network, memory management in the NAL protection domain, and concurrency control.

When a NAL is initialized, it does whatever is required to receive notifications about incoming data. For the TCP/IP NAL this means using a hook in the TCP/IP stack. For other networks, it may involve other libraries or a handshake with the NIC. When incoming data notifications occur, the NAL is responsible for reading data off the wire and giving it to Portals as requested. Once Portals has composed an outgoing packet, the NAL is asked to put the data on the network. Both of these are asynchronous mechanisms.

The Lustre team in collaboration with Lawrence Livermore National Laboratory has developed two Linux kernel NAL's, one using the Linux kernel socket interface and one using the Quadrics Software kernel communications library.

5. Acknowledgments

The authors would like to thank Lee Ward and Rumi Zahir for extensive discussions on Lustre networking and acknowledge the contributions of Arthur B. Maccabe and Rolf Riesen to the Portals architecture.

References

- [1] <http://www.lustre.org>.
- [2] <http://www.inter-mezzo.org>.
- [3] N. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. Myrinet-a gigabit-per-second local-area network. *IEEE Micro*, 15(1):29–36, February 1995.
- [4] R. B. Brightwell, L. A. Fisk, D. S. Greenberg, T. B. Hudson, M. J. Levenhagen, A. B. Maccabe, and R. E. Riesen. Massively Parallel Computing Using Commodity Components. *Parallel Computing*, 26(2-3):243–266, February 2000.
- [5] R. B. Brightwell, T. B. Hudson, A. B. Maccabe, and R. E. Riesen. The Portals 3.0 Message Passing Interface. Technical Report SAND99-2959, Sandia National Laboratories, December 1999.
- [6] G. A. Gibson, D. Nagle, K. Amiri, F. W. Chang, E. M. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, and J. Zelenka. File server scaling with network-attached secure disks. In *Measurement and Modeling of Computer Systems*, pages 272–284, 1997.
- [7] J. H. Howard. An overview of the andrew file system. In *Proceedings of the USENIX Winter Technical Conference*, 1988.
- [8] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. In *Thirteenth ACM Symposium on Operating Systems Principles*, volume 25 5, pages 213–225, Asilomar Conference Center, Pacific Grove, U.S., 1991. ACM Press.
- [9] A. B. Maccabe, K. S. McCurley, R. E. Riesen, and S. R. Wheat. SUNMOS for the Intel Paragon: A brief user's guide. In *Proceedings of the Intel Supercomputer Users' Group. 1994 Annual North America Users' Conference.*, pages 245–251, June 1994.
- [10] F. Petrini, W. chun Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics Network: High-Performance Clustering Technology. *IEEE Micro*, 22(1):46–57, Jan./Feb. 2002.
- [11] P. L. Shuler, C. Jong, R. E. Riesen, D. van Dresser, A. B. Maccabe, L. A. Fisk, and T. M. Stallcup. The Puma operating system for massively parallel computers. In *Proceedings of the 1995 Intel Supercomputer User's Group Conference*. Intel Supercomputer User's Group, 1995.