Lustre Protocol Documentation

Andrew Uselton <andrew.c.uselton@intel.com>

Revision History

Revision 0.0	January 2015	ACU

Table of Contents

The Lustre parallel file system provides a global POSIX namespace for the computing resources of a data center. Lustre runs on Linux-based hosts via kernel modules, and delegates block storage management to the back-end servers while providing object-based storage to its clients. Servers are responsible for both data objects (the contents of actual files) and index objects (for directory information). Data objects are gathered on Object Storage Servers (OSSs), and index objects are stored on MetaData Storage Servers (MDSs). Each back-end storage volume is a target with Object Storage Targets (OSTs) on OSSs, and MetaData Storage Targets (MDTs) on MDSs. Clients assemble the data from the MDT(s) and OST(s) to present a single coherent POSIX-compliant file system. The clients and servers communicate and coordinate among themselves via network protocols. A low-level protocol, LNet, abstracts the details of the underlying networking hardware and presents a uniform interface, originally based on Sandia Portals [PORTALS], to Lustre clients and servers. Lustre, in turn, layers its own protocol PtlRPC atop LNet. This document describes the Lustre protocols, including how they are implemented via PtlRPC and the Lustre Distributed Lock Manager (based on the VAX/VMS Distributed

Lock Manager). This document does not describe Lustre itself in any detail, except where doing so explains concepts that allow this document to be self-contained.

1. Introduction

Lustre runs across multiple hosts, coordinating the activities among those hosts via the exchange of messages over a network. On each host, Lustre is implemented via a collection of Linux processes (often called "threads"). This discussion will refer to a more formalized notion of *processes* that abstract some of the thread-level details. The description of the activities on each host comprise a collection of *abstract processes*. Each abstract process may be thought of as a state machine, or automaton, following a fixed set of rules for how it consumes messages, changes state, and produces other messages. The *behavior* of a process is shorthand for the management of its state and the rules governing what messages it can consume and produce. Processes communicate with each other via messages. The Lustre protocol is the collection of messages the processes exchange along with the rules governing the behavior of those processes.

In order to understand the Lustre protocol it is helpful to begin with a description of messages being exchanged. Lustre uses a particular format for its messages called PtlRPC. A PtlRPC message is a sequence of bytes in a particular order and with specific meaning associated with bytes in the message. The message (sequence of bytes) is delivered to a lower level communication mechanism called LNet in order to be transported from one host to another. This document will not discuss LNet beyond identifying it as a transport layer that abstracts any underlying details of the actual networking hardware.

The following discussion is intended to be self-contained, in that additional external documents are not necessary in order for one to understand (and indeed implement) the behaviors and messages described. Nevertheless, for the interested there will be occasional references directly into the Lustre code-base where one may see the protocol as it is realized in one particular implementation, that being Lustre-2.6.92-0 as pulled from the git repository for Lustre on January 26th, 2015. The sole exception to the rule that this document is self-contained is that the discussion will not be burdened by the actual numerical values for hard-coded implementation details like "magic" value numbers or flags and their fields. References to the source code will be provided as needed for a prospective (otherwise) black-box implementer to build a compatible implementation. This document will confine itself to the symbolic values.

2. Lustre Messages

The PtlRPC-based communication in Lustre is in the form of pairs of messages exchanged between hosts. One host sends a request message to another and awaits a reply from that other host. Leaving aside errors, lost messages, and host failures, this request and reply interaction forms the basis for implementing the Lustre protocol.

2.1. Lustre Operations

Each pair of messages corresponds to some useful operation and is given a name in the Lustre sources. For example MDS_CONNECT names a pair of messages whose operation is to establish a connection from the requesting host to a given MDS. This document will use those names (or easily recognized variants) to refer to the message pairs. The names are simply a convenience and have no direct role in the protocol. Note that the Lustre wireshark extension does use these names when reporting what messages are being exchanged. The "Message Pairs" appendix lists all the operations Lustre uses along with the names of the request and rely messages to carry out each operation.

The request and reply messages are PtlRPC messages. Each of Lustre’s PtlRPC-based messages is a sequence of bytes. It can vary in length and has additional structure, but its simplest expression is just a byte array. The bytes of a message can be divided into an initial "header" and one or more "buffers" that follow the header. This section ends with a detailed discussion of the header.

2.2. Message Formats

The sequence of buffers in a given message are arranged in a particular format, and there are several distinct message formats with each format given a symbolic name in the sources. For example the symbols for the two messages implementing the "MDS_CONNECT" operation are "obd_connect_client" and "obd_connect_server". For convenience, this document uses those names when referring to particular message formats. The "Message Formats" appendix details how Lustre’s messages are organized.

In many cases the host initiating a request will be a Lustre client, but that is not universally the case. For example, a lock call-back might be initiated by the MDS and the request sent from the MDS to a Lustre client. In other cases the message request is between two Lustre servers. Following the conventions in the Lustre source code, many of the message formats employ the words "client" or "server" in the format’s name. This can be misleading. Such "client" messages are not necessarily sent from Lustre clients, and such "server" messages are not necessarily replies sent from Lustre servers. A format with the word "client" should always be thought of simply as the format of a message initiating a request. Likewise, a "server" format simply means the format for a message in reply to a request, whatever the actual host that sends the reply.

2.3. Structures

Each message format names a list of structures, where each structure provides the content to be filled into the corresponding buffer. For example, the sequence of structures for the obd_connect_client format is: ptlrpc_body, obd_uuid, obd_uuid, lustre_handle, and obd_connect_data. A named structure, in turn, gives details about the sequence of bytes for that portion of the message. Most, but not all, of these named structures correspond directly to C *struct* definitions and can be found in the lustre/include/lustre directory in the files lustre_idl.h and lustre_user.h. In those cases the structure has a sequence of fields, which also have names and a semantics. In some cases the name does not correspond to a *struct*, but gives some other indication of its form: perhaps a *union* or an unsigned 32-bit integer. The complete list of named structures and their content are in the "Structures" appendix.

Every format begins with the structure "ptlrpc_body". That structure gives additional details that will assist the receiver with decoding the rest of the message. This includes, especially, the pb_opc field for the *op code* corresponding to the operation being requested.

Note especially that when a operation calls for a message format that is "empty", that does not mean that no request is sent or no reply expected. The "empty" format consists of a ptlrpc_body (together with the header) and nothing else.

2.4. Endianness

In a heterogeneous hardware environment it is possible that two hosts supporting or mounting the Lustre file system may differ in their "endianness". This would be true for "x86_64" hosts versus "powerpc" hosts. The convention for Lustre protocol messages is that they are encoded in the endian convention of the sender, that is, the host initiating the request and expecting a reply. When a message is received on a host and it is encoded in the "other" endianness convention, the byte order has to be swapped across the whole message before it can be decoded. That process is known as "swabbing".

2.5. The Message Header lustre_msg_v2

The "lustre_msg_v2" structure gives the sequence of fields in the header to a Lustre message. Its main task is to tell how many buffers will follow. The header is divided into a sequence of eight 4-byte "fields" (32-bit unsigned integers) followed by a variable length sequence of additional 4-byte entries organized as an array.

Table 1. lustre_msg_v2

type	field
__u32	lm_bufcount
__u32	lm_secflvr
__u32	lm_magic
__u32	lm_repsize
__u32	lm_cksum
__u32	lm_flags
__u32	lm_padding_2
__u32	lm_padding_3
__u32	lm_buflens[0]

lm_bufcount gives the number of buffers that will follow the header. The form and content of these buffers is determined by the message format.

lm_secflvr is an indication of whether any sort of cryptographic encoding of the subsequent buffers will be in force. The value is zero if there is no "crypto" and gives a code identifying the "flavor" of crypto if it is employed. Further, if crypto is employed there will only be one buffer following (i.e. bufcount = 1), and that buffer is an encoding of another full PtlRPC message with its own *struct lustre_msg_v2* header and sequence of buffers. This document will defer all discussion of cryptography. An addendum is planned that will address it separately.

lm_magic serves three purposes. First, the value must be recognizable as one of the possible "magic" values or the message is presumed to be corrupted. Second, the two recognizable values, LUSTRE_MSG_MAGIC_V1 and LUSTRE_MSG_MAGIC_V2 distinguish between the two possible Lustre protocol versions (and version 1 is obsolete). Third, the magic values asymmetric under byte ordering (big-endian encodings of them are different from little-endian encodings), so looking at the values reveals if the sender has a byte ordering convention different from the receiver. See the discussion on "swabbing".

lm_repsize in a reply is zero. In request it gives an indication of the maximum available space that has been set aside for the reply to the request. A reply that attempts to use more than that much space will be discarded by LNet.

lm_cksum is a checksum (CRC-32) of the header, including any padding (see below) but not including the additional buffers. The checksum is only used for early reply messages.

lm_flags is constructed by *or*-ing the two values MSGHDR_AT_SUPPORT and MSGHDR_CKSUM_INCOMPAT18. MSGHDR_AT_SUPPORT is set if the sender understands Adaptive Timeouts and can receive early replies for a request. MSGHDR_CKSUM_INCOMPAT18 is set if the lm_cksum field is computed for only the first 88 bytes (sizeof(lustre_msg_v1)) or the full sizeof(lustre_msg_v2).

lm_padding_2 and *lm_padding_3* are two 4-byte fields reserved for future use. Note also that the following array must be aligned on a 8-byte boundary, so the padding fields are included as a pair.

lm_buflens[] is an array of 4-byte unsigned integers with *lm_bufcount* entries. Each entry corresponds to, and gives the length of, one of the buffers that will follow and that constitute the remainder of the message. The length of the i^{th} buffer is given by the field *lm_buflens[i]*, and the buffers themselves follow any needed padding.

The first of the buffers following the header must be aligned on a 8-byte boundary. Since the length of the *lm_buflens* array is in increments of four bytes we may need four additional bytes of padding before the first buffer.

3. Shared State Between Clients and Servers

Shared state between clients and servers is implemented via the server "export" and the client "import" along with lock structures.

4. Namespace Operations

4.1. Mount

The *mount* operation is initiated on a client and requires Lustre services to already be in place (on the servers). In order to know what services are available the client must first find out from the MGS what the current configuration is.

Messages Between the Client and the MGS

In order to be able to mount the Lustre file system the client needs to know the identities of the various servers and targets so that it can initiate connections to them. More details to follow.

Messages Between the Client and the MDSs

Once the client knows about the MDS and its MDTs it begins connecting to the MDTs one-by-one. An MDT connect is carried out by the exchange of four message pairs. Each of the mesage pair carries out one operation. The four operations involved are named MDS_CONNECT, MDS_STATFS, MDS_GETSTATUS, and MDS_GETATTR, in that order. Note that while the names used (in the Lustre wireshark extension, for example) talk about MDS_CONNECT, etc., the operation is actually connecting to an MDT. If there are multiple MDTs the process is repeated for each one.

The MDS_CONNECT operation (see the "Message Pairs" appendix) is initiated by the client with the request message *obd_connect_client*, to which the server replies with an *obd_connect_server* message. The *obd_connect_client* message has a format (see the "Message Formats" appendix) that begins with the structure *ptlrpc_body*, followed by two *obd_uuid* structures, a *lustre_handle* structure, and finally an *obd_connect_data* structure. The first *obd_uuid* identifies the target and the second identifies the client. The *lustre_handle* holds a "cookie" that uniquely identifies this connection request, so the client can recognize the server’s reply and distinguish that reply from any other replies to connection requests.

The *ptlrpc_body* and *obd_connect_data* structures (see the "Structures" appendix) contain numerous fields that establish, for example, the (requested) capabilities for the file system to be mounted, its versions, and other properties.

The MDS_STATFS operation…

The MDS_GETSTATUS operation…

The MDS_GETATTR operation…

Messages Between the Client and the OSSs

Additional CONNECT message flow between the client and each OST enumerated by the MGS.

5. Data Movement Operations

Messages moving data between clients and object servers (OSSs) are the mechanism for performing bulk I/O in Lustre.

6. State Management

6.1. Connect

The client connect process…

Glossary

Here are some common terms used in discussing Lustre, POSIX semantics, and the protocols used to implement them.

Object Storage Server	An object storage server (OSS) is a computer responsible for running Lustre kernel services in support of managing bulk data objects on the underlying storage. There can be multiple OSSs in a Lustre file system.

MetaData Server	A metadata server (MDS) is a computer responsible for running the Lustre kernel services in support of managing the POSIX-compliant name space and the indices associating files in that name space with the locations of their corresponding objects. As of v2.4 there can be multiple MDSs in a Lustre file system.
Object Storage Target	An object storage target (OST) is the service provided by an OSS that mediates the placement of data objects on the specific underlying file system hardware. There can be multiple OSTs on a given OSS.
MetaData Target	A metadata target (MDT) is the service provided by an MDS that mediates the management of name space indices on the underlying file system hardware. As of v2.4 there can be multiple MDTs on an MDS.
server	A computer providing a service, such as an OSS or an MDS
target	Storage available to be served, such as an OST or an MDT. Also the service being provided.
protocol	An agreed upon formalism for communicating between two entities, such as between two servers or between a client and a server.
client	A computer taking advantage of a service provided by a server, such as a Lustre client using MDS(s) and OSS(s) to assemble a POSIX-compliant file system with its namespace and data storage capabilities.
PtlRPC	The protocol (or set of protocols) implemented via RPCs that is (are) employed by Lustre to communicate between its clients and servers.
Remote Procedure Call	A mechanism for implementing operations involving one computer acting on the behalf of another (RPC).
LNet	A lower level protocol employed by PtlRPC to abstract the mechanisms provided by various hardware centric protocols, such as TCP or Infiniband.

A. Structures

The structures listed here give the field names and sizes for portions of Lustre messages. The message formats list which structures go into each type message and in what order.

A.1. lu_fid

Table A.1. lu_fid

type	field
__u64	f_seq
__u32	f_oid
__u32	f_ver

A.2. lustre_handle

Table A.2. lustre_handle

type	field
__u64	cookie

A *struct lustre_handle* contains a single 64-bit field called a *cookie*. The cookie is used to identify shared state objects (import and exports, DLM locks, etc.) between clients and servers.

A.3. obd_connect_data

Table A.3. obd_connect_data

type	field
__u64	ocd_connect_flags
__u32	ocd_version
__u32	ocd_grant
__u32	ocd_index
__u32	ocd_brw_size
__u64	ocd_ibits_known
__u8	ocd_blocksize
__u8	ocd_inodespace
__u16	ocd_grant_extent
__u32	ocd_unused
__u64	ocd_transno
__u32	ocd_group
__u32	ocd_cksum_types
__u32	ocd_max_easize
__u32	ocd_instance
__u64	ocd_maxbytes
__u64	padding1
__u64	padding2
__u64	padding3
__u64	padding4
__u64	padding5
__u64	padding6
__u64	padding7
__u64	padding8

type	field
__u64	padding9
__u64	paddingA
__u64	paddingB
__u64	paddingC
__u64	paddingD
__u64	paddingE
__u64	paddingF

ocd_connect_flags are the flags the client and the server use to establish agreement about the services the target OBD can provide to the client. The client request message will propose a set of the flags and the server will reply with the subset it agrees to support for the given target. The complete list of flags is:

Table A.4. ocd_connect_flags

OBD_CONNECT_RDONLY
OBD_CONNECT_INDEX
OBD_CONNECT_MDS
OBD_CONNECT_GRANT
OBD_CONNECT_SRVLOCK
OBD_CONNECT_VERSION
OBD_CONNECT_REQPORTAL
OBD_CONNECT_ACL
OBD_CONNECT_XATTR
OBD_CONNECT_CROW
OBD_CONNECT_TRUNCLOCK
OBD_CONNECT_TRANSNO
OBD_CONNECT_IBITS
OBD_CONNECT_JOIN
OBD_CONNECT_ATTRFID
OBD_CONNECT_NODEVOH
OBD_CONNECT_RMT_CLIENT
OBD_CONNECT_RMT_CLIENT_FORCE
OBD_CONNECT_BRW_SIZE
OBD_CONNECT_QUOTA64
OBD_CONNECT_MDS_CAPA
OBD_CONNECT_OSS_CAPA
OBD_CONNECT_CANCELSET

OBD_CONNECT_SOM
OBD_CONNECT_AT
OBD_CONNECT_LRU_RESIZE
OBD_CONNECT_MDS_MDS
OBD_CONNECT_REAL
OBD_CONNECT_CHANGE_QS
OBD_CONNECT_CKSUM
OBD_CONNECT_FID
OBD_CONNECT_VBR
OBD_CONNECT_LOV_V3
OBD_CONNECT_GRANT_SHRINK
OBD_CONNECT_SKIP_ORPHAN
OBD_CONNECT_MAX_EASIZE
OBD_CONNECT_FULL20
OBD_CONNECT_LAYOUTLOCK
OBD_CONNECT_64BITHASH
OBD_CONNECT_MAXBYTES
OBD_CONNECT_IMP_RECOV
OBD_CONNECT_JOBSTATS
OBD_CONNECT_UMASK
OBD_CONNECT_EINPROGRESS
OBD_CONNECT_GRANT_PARAM
OBD_CONNECT_FLOCK_OWNER
OBD_CONNECT_LVB_TYPE
OBD_CONNECT_NANOSEC_TIME
OBD_CONNECT_LIGHTWEIGHT
OBD_CONNECT_SHORTIO
OBD_CONNECT_PINGLESS
OBD_CONNECT_FLOCK_DEAD
OBD_CONNECT_DISP_STRIPE
OBD_CONNECT_OPEN_BY_FID

ocd_version provides the Lustre version. A macro actually composes the value based on makor, minor, and release numbers.

ocd_grant will always be zero in a request. It is only used in replies for OSTs, where it will set the initial size of the grant for the client for that OST. The grant is a promise made by the OSS that data cached on the client, up to this amount, will be guaranteed to have a destination on the server when it comes time for the client to clear that cache. When the cache grant has been consumed on the client it must block further write requests. The grant can be increased again by subsequent messages from the server.

ocd_index provides the LOV index (for example the *0000* in lustrefs-OST0000) for a given target. This is only used for OSTs at this point. The MGS has already supplied both UUIDs and LOV indices for each target, so this value is a cross check that the client and the server both agree that the given UUID corresponds ot the same index.

ocd_brw_size is the size of the largest supported RPC. The client send a request with the largest it is willing to handle and the server replies with the smaller of the clients or its own value.

ocd_ibits_known is used to establish agreement between the client and the server about if and how locks can lock inode bits. The client request proposes a value and the server either agrees or further restricts it.

ocd_blocksize is only used in reply messages from the server. For a given target, this is the log-based-2 size of the fiel system.

ocd_inodespace is only used in reply messages from the server. this tells the client how big inodes are on the target.

ocd_grant_extent is only used in reply messages from the server. It also relates to client write-cache. In this case, there is (or may be) some overhead associated with writing each extent on the target. This is true, for example, in ZFS. The server informs the client of the extra overhead it must use in calculating how much of its grant has been consumed.

ocd_unused is unused.

ocd_transno is only used in replies from the server. It allows the server to inform the client about the last transaction the server had seen for the given target from that client.

ocd_group is used for MDS to OSS connections. With the advent of multiple MDTs on an MDS the MDS must keep track of the group (a backend filesystem directory on the OSS) being used to organize the name space of objects for a given OST.

ocd_cksum_types identifies the checksum methods a client can use in communication with an OSS. The client proposes the ones it is willing to use and the server selects the one it determines is best. The selected method is used to checksum data being sent between the client and the OSS.

ocd_max_easize is zero in a request. The server replies with the value appropriate to the given target. It governs the space being allocated in support of extended attibutes (EAs). Since stripe information is encoded in EAs it is an optimization to use less space if possible. If the target only supports or needs a limited stripe count, then the EA can be smaller than its maximum possible size.

ocd_instance is used only in replies by the MGS to other servers. It reports a value maintained on the MGS for the given server. As a server reconnects the MGS will increment this value (if appropriate). In imperative recovery the MGS can then proactively signal clients to reconnect to the server if needed.

ocd_maxbytes is only used in replies to a client request. It informs the client of the maximum size a stripe can grow to for the given target. This is essentially the size of the backend file system on the target.

padding1 and the rest are fields reserved for future use, but are not currently in use.

A.4. ost_id

Table A.5. ost_id

type	field
union __u64	oi_id, oi_seq
struct lu_fid	oi_fid

A.5. ptlrpc_body

The first buffer in every message format is always a *ptlrpc_body* structure.

Table A.6. ptlrpc_body

type	field
struct lustre_handle	pb_handle
__u32	pb_type
__u32	pb_version
__u32	pb_opc
__u32	pb_status
__u64	pb_last_xid
__u64	pb_last_seen
__u64	pb_last_committed
__u64	pb_transno
__u32	pb_flags
__u32	pb_op_flags
__u32	pb_conn_cnt
__u32	pb_timeout
__u32	pb_service_time
__u32	pb_limit
__u64	pb_slv
__u64	pb_pre_versions[PTLRPC_NUM_VERSIONS]
__u64	pb_padding[4]
char	pb_jobid[LUSTRE_JOBID_SIZE]

The semantics of each field will generally be different between request messages and replies.

pb_handle is a *struct lustre_handle* which holds a cookie. In a connection request (eg. MDS_CONNECT, from a client to a server) *pb_handle* is 0. In the reply to a connection request *pb_handle* will be the cookie that uniquely identifies the shared state information (the "export") for that client that is maintained on the server. The client then notes this cookie in its *import*. Subsequent messages between this client and this server will refer to the same shared state by using this cookie as the *lustre_handle* in this field.

pb_type is one of the three message types PTL_RPC_MSG_REQUEST, PTL_RPC_MSG_ERR, or PTL_RPC_MSG_REPL. As one might expect, "request" and "reply" are the two usual message types, one for initiating and exchange and the other for responding to it. The "err" message type is only for responding to a message that failed to be interpreted as an actual message. Note that other errors, such as those that emerge from processing the actual message content, do not use the PTL_RPC_MSG_ERR symbol.

pb_version is a field that encodes the Lustre protocol version (PTLRPC_MSG_VERSION) in combination (*or*-ed) with one of the service types: LUSTRE_OBD_VERSION, LUSTRE_MDS_VERSION, LUSTRE_OST_VERSION, LUSTRE_DLM_VERSION, LUSTRE_LOG_VERSION, or LUSTRE_MGS_VERSION.

pb_opc gives the actual operation that is the subject of this PtlRPC. The *op code* for an operation is one of the names from the list of message pairs, for example MDS_CONNECT.

pb_status will always be zero in a request message. In a reply message, a zero indicates that the service successfully initiated the requested operation. If for some reason the operation could not be initiated (eg. "permission denied") *pb_status* will encode the standard Linux kernel (POSIX) error code (eg. EPERM). Note the difference between *pb_status* errors, where there is a problem in the processing of an otherwise well-formed message, and errors that come from the message itself being ill-formed (which results in a pb_type=RPC_MSG_ERR). Note also that a *pb_status* of zero returned for an operation does not necessarily mean it has completed (cf. pb_last_committed).

pb_last_xid is not used.

pb_last_seen is not used.

pb_last_committed is always zero in a request. In a reply it is the highest transaction number that has been committed to storage. The transaction numbers are maintained on a per-target basis and each series of transaction numbers is a monotonically increasing sequence. This field is set in any kind of reply message including pings and non-modifying transactions.

pb_transno always zero in a request. It is also zero for replies to operations that do not modify the file system. For replies to operations that do modify the file system it is the server-assigned value from the monotonically increasing sequence of 64-bit values associated with the given client.

pb_flags is one among: MSG_LAST_REPLAY, MSG_RESENT, MSG_REPLAY, MSG_DELAY_REPLAY, MSG_VERSION_REPLAY, MSG_REQ_REPLAY_DONE, and MSG_LOCK_REPLAY_DONE.

pb_op_flags is one among: MSG_CONNECT_RECOVERING, MSG_CONNECT_RECONNECT, MSG_CONNECT_REPLAYABLE, MSG_CONNECT_LIBCLIENT, MSG_CONNECT_INITIAL, MSG_CONNECT_ASYNC, MSG_CONNECT_NEXT_VER, and MSG_CONNECT_TRANSNO.

pb_conn_cnt in a request message reports the client’s *era*, which is part of the client and server’s shared state. The value of the *era* is initialized to one when it is first connected to the metadata service. Each subsequent connection (after an eviction) increments the *era*. Since the *pb_conn_cnt* reflects the client’s *era* at the time the message was composed the server can use this value to discard late-arriving messages requesting operations on out-of-date shared state. *fixme* Will the server also send a current *era* or will replies just be zero?

pb_timeout in a request indicates how long the requester plans to wait before timing out the operation. That is, the corresponding reply for this message should arrive within this time-frame. The service may extend this time-frame via an *early reply*, which is a reply to this message that notifies the requester that it should extend its timeout interval by the value of *pb_time* (the one in the reply). The *early reply* does not indicate the operation has actually been initiated. Clients maintain multiple request queues, called "portals", and each type of operation is assigned to one of these queues. There is a timeout value associated with each queue, and the *pb_timeout* update affects all the messages associated with the given queue, not just the specific message that initiated the request. Finally, in a reply message (one that does indicate the operation has been initiated) the *pb_timeout* value updates the timeout interval for the queue.

pb_service_time is zero in a request. In a reply it indicates how long this particular operation actually took from the time it first arrived in the request queue (at the service) to the time the server replied. Note that the client can use this value and the local elapsed time for the operation to calculate network latency.

pb_limit is zero in a request. In a reply it is a value sent from a lock service to a client to set the maximum number of locks available to the client. When dynamic lock LRU's are enabled this allows for managing the size of the LRU.

pb_slv is zero in a request. The "server lock volume" on a DLM service is a value that characterizes (estimates) the amount of traffic, or load, on that lock service. It is calculated as the product of the number of locks and their age. In a reply, the *pb_slv* value indicates to the client the available share of the total lock load on the server that it is allowed to consume. The client is then responsible for reducing its number or(or age) of locks to stay within this limit.

pb_pre_versions[PTLRPC_NUM_VERSIONS] has four entries (PTLRPC_NUM_VERSIONS = 4). They are always zero in a request message. They are also zero in replies to operations that do not modify the file system. For an operation that does modify the file system, the reply encodes the most recent transaction numbers for the objects modified by this operation.

pb_padding[4] is reserved for use.

pb_jobid[LUSTRE_JOBID_SIZE] gives a unique identifier associated with the process on behalf of which this message was generated. The identifier is assigned to the user process by a job scheduler, if any.

B. Message Formats

This table lists the names of the message formats employed in Lustre messages along with the structures they employ. These formats are gathered into message pairs where one format in the pair is the format for a request message and the other is the format for its anticipated reply.

Table B.1. message formats and their structures

format	structures
empty	ptlrpc_body
fld_query_client	ptlrpc_body
	fld_query_opc(__u32)

format	**structures**
	lu_seq_range
fld_query_server	ptlrpc_body
	lu_seq_range
fld_read_client	ptlrpc_body
	lu_seq_range
fld_read_server	ptlrpc_body
	unstructured data
ldlm_cp_callback_client	ptlrpc_body
	ldlm_request
	unstructured data
ldlm_enqueue_client	ptlrpc_body
	ldlm_request
ldlm_enqueue_lvb_server	ptlrpc_body
	ldlm_reply
	unstructured data
ldlm_enqueue_server	ptlrpc_body
	ldlm_reply
ldlm_gl_callback_desc_client	ptlrpc_body
	ldlm_request
	ldlm_gl_desc
ldlm_gl_callback_server	ptlrpc_body
	unstructured data
ldlm_intent_basic_client	ptlrpc_body
	ldlm_request
	ldlm_intent
ldlm_intent_client	ptlrpc_body
	ldlm_request
	ldlm_intent
	mdt_rec_reint
ldlm_intent_create_client	ptlrpc_body
	ldlm_request
	ldlm_intent
	mdt_rec_reint
	lustre_capa
	unstructured data

format	**structures**
	unstructured data
ldlm_intent_getattr_client	ptlrpc_body
	ldlm_request
	ldlm_intent
	mdt_body
	lustre_capa
	unstructured data
ldlm_intent_getattr_server	ptlrpc_body
	ldlm_reply
	mdt_body
	MIN_MD_SIZE
	acl
	lustre_capa
ldlm_intent_getxattr_client	ptlrpc_body
	ldlm_request
	ldlm_intent
	mdt_body
	lustre_capa
ldlm_intent_getxattr_server	ptlrpc_body
	ldlm_reply
	mdt_body
	MIN_MD_SIZE
	acl
	unstructured data
	unstructured data
	unstructured data
ldlm_intent_layout_client	ptlrpc_body
	ldlm_request
	ldlm_intent
	layout_intent
	unstructured data
ldlm_intent_open_client	ptlrpc_body
	ldlm_request
	ldlm_intent
	mdt_rec_reint

format	structures
	lustre_capa
	lustre_capa
	unstructured data
	unstructured data
ldlm_intent_open_server	ptlrpc_body
	ldlm_reply
	mdt_body
	MIN_MD_SIZE
	acl
	lustre_capa
	lustre_capa
ldlm_intent_quota_client	ptlrpc_body
	ldlm_request
	ldlm_intent
	quota_body
ldlm_intent_quota_server	ptlrpc_body
	ldlm_reply
	unstructured data
	quota_body
ldlm_intent_server	ptlrpc_body
	ldlm_reply
	mdt_body
	MIN_MD_SIZE
	acl
ldlm_intent_unlink_client	ptlrpc_body
	ldlm_request
	ldlm_intent
	mdt_rec_reint
	lustre_capa
	unstructured data
llog_log_hdr_only	ptlrpc_body
	llog_log_hdr
llog_origin_handle_create_client	ptlrpc_body
	llogd_body
	unstructured data

format	**structures**
llog_origin_handle_next_block_server	ptlrpc_body
	llogd_body
	unstructured data
llogd_body_only	ptlrpc_body
	llogd_body
llogd_conn_body_only	ptlrpc_body
	llogd_conn_body
log_cancel_client	ptlrpc_body
	llog_cookie
mds_getattr_name_client	ptlrpc_body
	mdt_body
	lustre_capa
	unstructured data
mds_getattr_server	ptlrpc_body
	mdt_body
	MIN_MD_SIZE
	acl
	lustre_capa
	lustre_capa
mds_getinfo_client	ptlrpc_body
	unstructured data
	getinfo_vallen(__u32)
mds_getinfo_server	ptlrpc_body
	unstructured data
mds_getxattr_client	ptlrpc_body
	mdt_body
	lustre_capa
	unstructured data
	unstructured data
mds_getxattr_server	ptlrpc_body
	mdt_body
	unstructured data
mds_last_unlink_server	ptlrpc_body
	mdt_body
	MIN_MD_SIZE

format	**structures**
	llog_cookie
	lustre_capa
	lustre_capa
mds_reint_client	ptlrpc_body
	mdt_rec_reint
mds_reint_create_client	ptlrpc_body
	mdt_rec_reint
	lustre_capa
	unstructured data
mds_reint_create_rmt_acl_client	ptlrpc_body
	mdt_rec_reint
	lustre_capa
	unstructured data
	unstructured data
	ldlm_request
mds_reint_create_slave_client	ptlrpc_body
	mdt_rec_reint
	lustre_capa
	unstructured data
	unstructured data
	ldlm_request
mds_reint_create_sym_client	ptlrpc_body
	mdt_rec_reint
	lustre_capa
	unstructured data
	unstructured data
	ldlm_request
mds_reint_link_client	ptlrpc_body
	RMF_REC_REIN
	lustre_capa
	lustre_capa
	unstructured data
	ldlm_request
mds_reint_open_client	ptlrpc_body
	mdt_rec_reint

format	**structures**
	lustre_capa
	lustre_capa
	unstructured data
	unstructured data
mds_reint_open_server	ptlrpc_body
	mdt_body
	MIN_MD_SIZE
	acl
	lustre_capa
	lustre_capa
mds_reint_rename_client	ptlrpc_body
	mdt_rec_reint
	lustre_capa
	lustre_capa
	unstructured data
	unstructured data
	ldlm_request
mds_reint_setattr_client	ptlrpc_body
	mdt_rec_reint
	lustre_capa
	mdt_ioepoch
	unstructured data
	llog_cookie
	ldlm_request
mds_reint_setxattr_client	ptlrpc_body
	mdt_rec_reint
	lustre_capa
	unstructured data
	unstructured data
	ldlm_request
mds_reint_unlink_client	ptlrpc_body
	mdt_rec_reint
	lustre_capa
	unstructured data
	ldlm_request

format	**structures**
mds_setattr_server	ptlrpc_body
	mdt_body
	MIN_MD_SIZE
	acl
	lustre_capa
	lustre_capa
mds_update_client	ptlrpc_body
	unstructured data
mds_update_server	ptlrpc_body
	unstructured data
mdt_body_capa	ptlrpc_body
	mdt_body
	lustre_capa
mdt_body_only	ptlrpc_body
	mdt_body
mdt_close_client	ptlrpc_body
	mdt_ioepoch
	mdt_rec_reint
	lustre_capa
mdt_hsm_action_server	ptlrpc_body
	mdt_body
mdt_hsm_ct_register	ptlrpc_body
	mdt_body
	hsm_archive(__u32)
mdt_hsm_ct_unregister	ptlrpc_body
	mdt_body
mdt_hsm_progress	ptlrpc_body
	mdt_body
	hsm_progress_kernel
mdt_hsm_request	ptlrpc_body
	mdt_body
	hsm_request
	hsm_user_item
	unstructured data
mdt_hsm_state_get_server	ptlrpc_body

format	structures
	mdt_body
	hsm_user_state
mdt_hsm_state_set	ptlrpc_body
	mdt_body
	lustre_capa
	hsm_state_set
mdt_release_close_client	ptlrpc_body
	mdt_ioepoch
	mdt_rec_reint
	lustre_capa
	close_data
mdt_swap_layouts	ptlrpc_body
	mdt_body
	mdc_swap_layouts
	lustre_capa
	lustre_capa
	ldlm_request
mgs_config_read_client	ptlrpc_body
	mgs_config_body
mgs_config_read_server	ptlrpc_body
	mgs_config_res
mgs_set_info	ptlrpc_body
	mgs_send_param
mgs_target_info_only	ptlrpc_body
	mgs_target_info
obd_connect_client	ptlrpc_body
	obd_uuid
	obd_uuid
	lustre_handle
	obd_connect_data
obd_connect_server	ptlrpc_body
	obd_connect_data
obd_idx_read_client	ptlrpc_body
	idx_info
obd_idx_read_server	ptlrpc_body

format	**structures**
	idx_info
obd_lfsck_reply	ptlrpc_body
	lfsck_reply
obd_lfsck_request	ptlrpc_body
	lfsck_request
obd_set_info_client	ptlrpc_body
	unstructured data
	unstructured data
obd_statfs_server	ptlrpc_body
	obd_statfs
ost_body_capa	ptlrpc_body
	ost_body
	lustre_capa
ost_body_only	ptlrpc_body
	ost_body
ost_brw_client	ptlrpc_body
	ost_body
	obd_ioobj
	niobuf_remote
	lustre_capa
ost_brw_read_server	ptlrpc_body
	ost_body
ost_brw_write_server	ptlrpc_body
	ost_body
	niobuf_remote (__u32)
ost_destroy_client	ptlrpc_body
	ost_body
	ldlm_request
	lustre_capa
ost_get_fiemap_client	ptlrpc_body
	ll_fiemap_info_key
	unstructured data
ost_get_fiemap_server	ptlrpc_body
	unstructured data
ost_get_info_generic_client	ptlrpc_body

format	structures
	unstructured data
ost_get_info_generic_server	ptlrpc_body
	unstructured data
ost_get_last_fid_client	ptlrpc_body
	unstructured data
	lu_fid
ost_get_last_fid_server	ptlrpc_body
	lu_fid
ost_get_last_id_server	ptlrpc_body
	obd_id (__u64)
ost_grant_shrink_client	ptlrpc_body
	unstructured data
	ost_body
quota_body_only	ptlrpc_body
	quota_body
quotactl_only	ptlrpc_body
	obd_quotactl
seq_query_client	ptlrpc_body
	seq_query_opc (__u32)
	lu_seq_range
seq_query_server	ptlrpc_body
	lu_seq_range

C. Message Pairs

Each message pair has a name and two message formats. The message pair’s name is a convenience to facilitate discussion. One may think of the name as an operation Lustre will carry out. The two message formats in a message pair are the request message and a reply message appropriate to that request. Each of these messages is given a name in order to identify its specific format. Those names are reused so that there are 95 named messages formats to be used in the 94 message pairs.

Table C.1. Operations and their message pairs

Operation	Request Format	Reply Format
CONNECT	obd_connect_client	obd_connect_server
FLD_QUERY	fld_query_client	fld_query_server
FLD_READ	fld_read_client	fld_read_server
LDLM_BL_CALLBACK	ldlm_enqueue_client	empty

Operation	**Request Format**	**Reply Format**
LDLM_CALLBACK	ldlm_enqueue_client	empty
LDLM_CANCEL	ldlm_enqueue_client	empty
LDLM_CONVERT	ldlm_enqueue_client	ldlm_enqueue_server
LDLM_CP_CALLBACK	ldlm_cp_callback_client	empty
LDLM_ENQUEUE	ldlm_enqueue_client	ldlm_enqueue_lvb_server
LDLM_ENQUEUE_LVB	ldlm_enqueue_client	ldlm_enqueue_lvb_server
LDLM_GL_CALLBACK	ldlm_enqueue_client	ldlm_gl_callback_server
LDLM_GL_DESC_CALLBACK	ldlm_gl_callback_desc_client	ldlm_gl_callback_server
LDLM_INTENT	ldlm_intent_client	ldlm_intent_server
LDLM_INTENT_BASIC	ldlm_intent_basic_client	ldlm_enqueue_lvb_server
LDLM_INTENT_CREATE	ldlm_intent_create_client	ldlm_intent_getattr_server
LDLM_INTENT_GETATTR	ldlm_intent_getattr_client	ldlm_intent_getattr_server
LDLM_INTENT_GETXATTR	ldlm_intent_getxattr_client	ldlm_intent_getxattr_server
LDLM_INTENT_LAYOUT	ldlm_intent_layout_client	ldlm_enqueue_lvb_server
LDLM_INTENT_OPEN	ldlm_intent_open_client	ldlm_intent_open_server
LDLM_INTENT_QUOTA	ldlm_intent_quota_client	ldlm_intent_quota_server
LDLM_INTENT_UNLINK	ldlm_intent_unlink_client	ldlm_intent_server
LFSCK_NOTIFY	obd_lfsck_request	empty
LFSCK_QUERY	obd_lfsck_request	obd_lfsck_reply
LLOG_ORIGIN_CONNECT	llogd_conn_body_only	empty
LLOG_ORIGIN_HANDLE_CREATE	llog_origin_handle_create_client	llogd_body_only
LLOG_ORIGIN_HANDLE_DESTROY	llogd_body_only	llogd_body_only
LLOG_ORIGIN_HANDLE_NEXT_BLOCK	llogd_body_only	llog_origin_handle_next_block_server
LLOG_ORIGIN_HANDLE_PREV_BLOCK	llogd_body_only	llog_origin_handle_next_block_server
LLOG_ORIGIN_HANDLE_READ_HEADER	llogd_body_only	llog_log_hdr_only
LOG_CANCEL	log_cancel_client	empty
MDS_CLOSE	mdt_close_client	mds_last_unlink_server
MDS_CONNECT	obd_connect_client	obd_connect_server
MDS_DISCONNECT	empty	empty
MDS_DONE_WRITING	mdt_close_client	mdt_body_only
MDS_GETATTR	mdt_body_capa	mds_getattr_server
MDS_GETATTR_NAME	mds_getattr_name_client	mds_getattr_server
MDS_GETSTATUS	mdt_body_only	mdt_body_capa
MDS_GETXATTR	mds_getxattr_client	mds_getxattr_server
MDS_GET_INFO	mds_getinfo_client	mds_getinfo_server

Operation	**Request Format**	**Reply Format**
MDS_HSM_ACTION	mdt_body_capa	mdt_hsm_action_server
MDS_HSM_CT_REGISTER	mdt_hsm_ct_register	empty
MDS_HSM_CT_UNREGISTER	mdt_hsm_ct_unregister	empty
MDS_HSM_PROGRESS	mdt_hsm_progress	empty
MDS_HSM_REQUEST	mdt_hsm_request	empty
MDS_HSM_STATE_GET	mdt_body_capa	mdt_hsm_state_get_server
MDS_HSM_STATE_SET	mdt_hsm_state_set	empty
MDS_QUOTACHECK	quotactl_only	empty
MDS_QUOTACTL	quotactl_only	quotactl_only
MDS_READPAGE	mdt_body_capa	mdt_body_only
MDS_REINT	mds_reint_client	mdt_body_only
MDS_REINT_CREATE	mds_reint_create_client	mdt_body_capa
MDS_REINT_CREATE_RMT_ACL	mds_reint_create_rmt_acl_client	mdt_body_capa
MDS_REINT_CREATE_SLAVE	mds_reint_create_slave_client	mdt_body_capa
MDS_REINT_CREATE_SYM	mds_reint_create_sym_client	mdt_body_capa
MDS_REINT_LINK	mds_reint_link_client	mdt_body_only
MDS_REINT_OPEN	mds_reint_open_client	mds_reint_open_server
MDS_REINT_RENAME	mds_reint_rename_client	mds_last_unlink_server
MDS_REINT_SETATTR	mds_reint_setattr_client	mds_setattr_server
MDS_REINT_SETXATTR	mds_reint_setxattr_client	mdt_body_only
MDS_REINT_UNLINK	mds_reint_unlink_client	mds_last_unlink_server
MDS_RELEASE_CLOSE	mdt_release_close_client	mds_last_unlink_server
MDS_STATFS	empty	obd_statfs_server
MDS_SWAP_LAYOUTS	mdt_swap_layouts	empty
MDS_SYNC	mdt_body_capa	mdt_body_only
MGS_CONFIG_READ	mgs_config_read_client	mgs_config_read_server
MGS_SET_INFO	mgs_set_info	mgs_set_info
MGS_TARGET_REG	mgs_target_info_only	mgs_target_info_only
OBD_IDX_READ	obd_idx_read_client	obd_idx_read_server
OBD_PING	empty	empty
OBD_SET_INFO	obd_set_info_client	empty
OST_BRW_READ	ost_brw_client	ost_brw_read_server
OST_BRW_WRITE	ost_brw_client	ost_brw_write_server
OST_CONNECT	obd_connect_client	obd_connect_server
OST_CREATE	ost_body_only	ost_body_only

Operation	Request Format	Reply Format
OST_DESTROY	ost_destroy_client	ost_body_only
OST_DISCONNECT	empty	empty
OST_GETATTR	ost_body_capa	ost_body_only
OST_GET_INFO	ost_get_info_generic_client	ost_get_info_generic_server
OST_GET_INFO_FIEMAP	ost_get_fiemap_client	ost_get_fiemap_server
OST_GET_INFO_LAST_FID	ost_get_last_fid_client	ost_get_last_fid_server
OST_GET_INFO_LAST_ID	ost_get_info_generic_client	ost_get_last_id_server
OST_PUNCH	ost_body_capa	ost_body_only
OST_QUOTACHECK	quotactl_only	empty
OST_QUOTACTL	quotactl_only	quotactl_only
OST_SETATTR	ost_body_capa	ost_body_only
OST_SET_GRANT_INFO	ost_grant_shrink_client	ost_body_only
OST_SET_INFO_LAST_FID	obd_set_info_client	empty
OST_STATFS	empty	obd_statfs_server
OST_SYNC	ost_body_capa	ost_body_only
OUT_UPDATE	mds_update_client	mds_update_server
QC_CALLBACK	quotactl_only	empty
QUOTA_DQACQ	quota_body_only	quota_body_only
SEC_CTX	empty	empty
SEQ_QUERY	seq_query_client	seq_query_server

D. Concepts

Content to be provided

E. License

Bibliography

Here is a selected list of references, including those cited in the foregoing text.

[lustre] *Lustre*. http://lustre.opensfs.org

[POSIX] *POSIX*. http://pubs.opengroup.org/onlinepubs/9699919799/

[PORTALS] *The Portals 3.0 Message Passing Interface Revision 1.1.*. Ron Brightwell, Trammel Hudson, Rolf Riesen, and Arthur B. Maccabe. Technical report, December 1999.

[VAX_DLM] *The VAX/VMS Distributed Lock Manager*. W Snaman and D Thiel. Digital Technical Journal, September 1987.

[Barton_Dilger] *Lustre*. Eric Barton and Andreas Dilger. A book on parallel file systems. Chapter 8. High Performance Parallel I/O, Prabhat and Quincey Koziol, Chapman and Hall/CRC Press, 2014, ISBN: 978-1466582347.