Lustre™ 1.8 Operations Manual

Sun Microsystems, Inc.
www.sun.com

Part No. 821-0035-11

Lustre manual version: Lustre_1.8_man_v1.3

February 2010

Please
Recycle

Please
Recycle

Contents

Preface

The *Lustre 1.8 Operations Manual* provides detailed information and procedures to install, configure and tune Lustre. The manual covers topics such as failover, quotas, striping and bonding. The Lustre manual also contains troubleshooting information and tips to improve Lustre operation and performance.

Using UNIX Commands

This document might not contain information about basic UNIX® commands and procedures such as shutting down the system, booting the system, and configuring devices. Refer to the following for this information:

- Software documentation that you received with your system
- Solaris™ Operating System documentation, which is at:

  http://docs.sun.com

Shell Prompts

Shell	Prompt
C shell	*machine-name*%
C shell superuser	*machine-name*#
Bourne shell and Korn shell	$
Bourne shell and Korn shell superuser	#

Typographic Conventions

Typeface	Meaning	Examples
AaBbCc123	The names of commands, files, and directories; on-screen computer output	Edit your .login file. Use ls -a to list all files. % You have mail.
AaBbCc123	What you type, when contrasted with on-screen computer output	% **su** Password:
AaBbCc123	Book titles, new words or terms, words to be emphasized. Replace command-line variables with real names or values.	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this. To delete a file, type rm *filename*.

Note – Characters display differently depending on browser settings. If characters do not display correctly, change the character encoding in your browser to Unicode UTF-8.

A '\' (backslash) continuation character is used to indicate that commands are too long to fit on one text line.

Third-Party Web Sites

Sun is not responsible for the availability of third-party web sites mentioned in this document. Sun does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Sun will not be responsible or liable for any actual or alleged damage or loss caused by or in connection with the use of or reliance on any such content, goods, or services that are available on or through such sites or resources.

Revision History

BookTitle	Part Number	Rev	Date	Comments
Lustre 1.8 Operations Manual	821-0035-11	A	February 2010	First release of Lustre 1.8 manual
Lustre 1.8 Operations Manual	821-0035-11	B	October 2009	Second release of Lustre 1.8 manual
Lustre 1.8 Operations Manual	821-0035-11	c	February 2010	Third release of Lustre 1.8 manual

PART I Lustre Architecture

Lustre is a storage-architecture for clusters. The central component is the Lustre file system, a shared file system for clusters. The Lustre file system is currently available for Linux and provides a POSIX-compliant UNIX file system interface.

The Lustre architecture is used for many different kinds of clusters. It is best known for powering seven of the ten largest high-performance computing (HPC) clusters in the world with tens of thousands of client systems, petabytes (PBs) of storage and hundreds of gigabytes per second (GB/sec) of I/O throughput. Many HPC sites use Lustre as a site-wide global file system, servicing dozens of clusters on an unprecedented scale.

CHAPTER 1

Introduction to Lustre

This chapter describes Lustre software and components, and includes the following sections:

- Introducing the Lustre File System
- Lustre Components
- Lustre Systems
- Files in the Lustre File System
- Lustre Configurations
- Lustre Networking
- Lustre Failover and Rolling Upgrades

These instructions assume you have some familiarity with Linux system administration, cluster systems and network technologies.

1.1 Introducing the Lustre File System

Lustre is a storage architecture for clusters. The central component is the Lustre file system, which is available for Linux and provides a POSIX-compliant UNIX file system interface.

The Lustre architecture is used for many different kinds of clusters. It is best known for powering seven of the ten largest high-performance computing (HPC) clusters worldwide, with tens of thousands of client systems, petabytes (PB) of storage and hundreds of gigabytes per second (GB/sec) of I/O throughput. Many HPC sites use Lustre as a site-wide global file system, serving dozens of clusters on an unprecedented scale.

The scalability of a Lustre file system reduces the need to deploy many separate file systems (such as one for each cluster). This offers significant storage management advantages, for example, avoiding maintenance of multiple data copies staged on multiple file systems. Hand in hand with aggregating file system capacity with many servers, I/O throughput is also aggregated and scales with additional servers. Moreover, throughput (or capacity) can be easily adjusted by adding servers dynamically.

Lustre has been integrated with several vendor's kernels. We offer Red Hat Enterprise Linux (RHEL) and SUSE Linux Enterprise (SUSE) kernels with Lustre patches.

1.1.1 Lustre Key Features

The key features of Lustre include:

- **Scalability**: Lustre scales up or down with respect to the number of client nodes, disk storage and bandwidth. Currently, Lustre is running in production environments with up to 26,000 client nodes, with many clusters in the 10,000-20,000 client range. Other Lustre installations provide aggregated disk storage and bandwidth of up to 1,000 OSTs running on more than 450 OSSs. Several Lustre file systems with a capacity of 1 PB or more (allowing storage of up to 2 billion files) have been in use since 2006.
- **Performance**: Lustre deployments in production environments currently offer performance of up to 100 GB/s. In a test environment, a performance of 130 GB/s and 13,000 creates/s has been sustained. Lustre single client node throughput has been measured at 2 GB/s (max) and OSS throughput at 2.5 GB/s (max). Lustre has been run at 240 GB/sec on the Spider file system at Oak Ridge National Laboratories.
- **POSIX compliance**: The full POSIX test suite passes on Lustre clients. In a cluster, POSIX compliance means that most operations are atomic and clients never see stale data or metadata.
- **High-availability**: Lustre offers shared storage partitions for OSS targets (OSTs), and a shared storage partition for the MDS target (MDT).
- **Security**: In Lustre, it is an option to have TCP connections only from privileged ports. Group membership handling is server-based. POSIX access control lists (ACLs) are supported.
- **Open source**: Lustre is licensed under the GNU GPL.

Additionally, Lustre offers these features:

- **Interoperability**: Lustre runs on a variety of CPU architectures and mixed-endian clusters and interoperability between adjacent Lustre software releases.
- **Access control list (ACL)**: Currently, the Lustre security model follows a UNIX file system, enhanced with POSIX ACLs. Noteworthy additional features include root squash and connecting from privileged ports only.
- **Quotas**: User and group quotas are available for Lustre.
- **OSS addition**: The capacity of a Lustre file system and aggregate cluster bandwidth can be increased without interrupting any operations by adding a new OSS with OSTs to the cluster.
- **Controlled striping**: The default stripe count and stripe size can be controlled in various ways. The file system has a default setting that is determined at format time. Directories can be given an attribute so that all files under that directory (and recursively under any sub-directory) have a striping pattern determined by the attribute. Finally, utilities and application libraries are provided to control the striping of an individual file at creation time.

- **Snapshots**: Lustre file servers use volumes attached to the server nodes. The Lustre software includes a utility (using LVM snapshot technology) to create a snapshot of all volumes and group snapshots together in a snapshot file system that can be mounted with Lustre.
- **Backup tools**: Lustre 1.6 includes two utilities supporting backups. One tool scans file systems and locates files modified since a certain timeframe. This utility makes modified files' pathnames available so they can be processed in parallel by other utilities (such as rsync) using multiple clients. Another useful tool is a modified version of GNU tar (gtar) which can back up and restore extended attributes (i.e. file striping and pool membership) for Lustre.[1]
- Other current features of Lustre are described in detail in this manual. Future features are described in the Lustre roadmap.

1. Files backed up using the modified version of gtar are restored per the backed up striping information. The backup procedure does not use default striping rules.

1.2 Lustre Components

A Lustre file system consists of the following basic components (see FIGURE 1-1).

- **Metadata Server (MDS)** - The MDS server makes metadata stored in one or more MDTs available to Lustre clients. Each MDS manages the names and directories in the Lustre file system(s) and provides network request handling for one or more local MDTs.
- **Metadata Target (MDT)** - The MDT stores metadata (such as filenames, directories, permissions and file layout) on an MDS. Each file system has one MDT. An MDT on a shared storage target can be available to many MDSs, although only one should actually use it. If an active MDS fails, a passive MDS can serve the MDT and make it available to clients. This is referred to as MDS failover.
- **Object Storage Servers (OSS)**: The OSS provides file I/O service, and network request handling for one or more local OSTs. Typically, an OSS serves between 2 and 8 OSTs, up to 8 TB each[2]. The MDT, OSTs and Lustre clients can run concurrently (in any mixture) on a single node. However, a typical configuration is an MDT on a dedicated node, two or more OSTs on each OSS node, and a client on each of a large number of compute nodes.
- **Object Storage Target (OST)**: The OST stores file data (chunks of user files) as data objects on one or more OSSs. A single Lustre file system can have multiple OSTs, each serving a subset of file data. There is not necessarily a 1:1 correspondence between a file and an OST. To optimize performance, a file may be spread over many OSTs. A Logical Object Volume (LOV), manages file striping across many OSTs.
- **Lustre clients**: Lustre clients are computational, visualization or desktop nodes that run Lustre software that allows them to mount the Lustre file system.

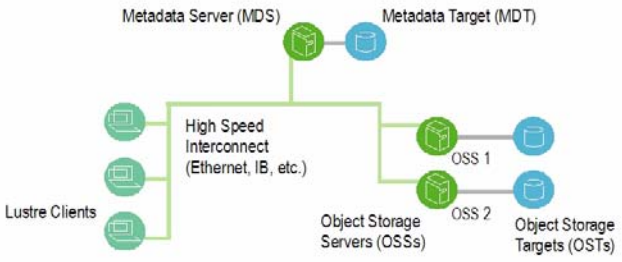
  The Lustre client software consists of an interface between the Linux Virtual File System and the Lustre servers. Each target has a client counterpart: Metadata Client (MDC), Object Storage Client (OSC), and a Management Client (MGC). A group of OSCs are wrapped into a single LOV. Working in concert, the OSCs provide transparent access to the file system.

  Clients, which mount the Lustre file system, see a single, coherent, synchronized namespace at all times. Different clients can write to different parts of the same file at the same time, while other clients can read from the file.

Lustre includes several additional components, LNET and the MGS, described in the following sections.

2. In Lustre 1.8.2, 16 TB OSTs are supported on RHEL 5 using specific RPMs (with ext4-based ldiskfs).

FIGURE 1-1 Lustre components in a basic cluster



1.2.1 Lustre Networking (LNET)

Lustre Networking (LNET) is an API that handles metadata and file I/O data for file system servers and clients. LNET supports multiple, heterogeneous interfaces on clients and servers. LNET interoperates with a variety of network transports through Network Abstraction Layers (NAL). Lustre Network Drivers (LNDs) are available for a number of commodity and high-end networks, including Infiniband, TCP/IP, Quadrics Elan, Myrinet (MX and GM) and Cray.

In clusters with a Lustre file system, servers and clients communicate with one another over a custom networking API known as Lustre Networking (LNET), while the disk storage behind the MDSs and OSSs is connected to these servers using traditional SAN technologies.

Key features of LNET include:

- RDMA, when supported by underlying networks such as Elan, Myrinet and InfiniBand.
- Support for many commonly-used network types such as InfiniBand and IP.
- High availability and recovery features enabling transparent recovery in conjunction with failover servers.
- Simultaneous availability of multiple network types with routing between them.

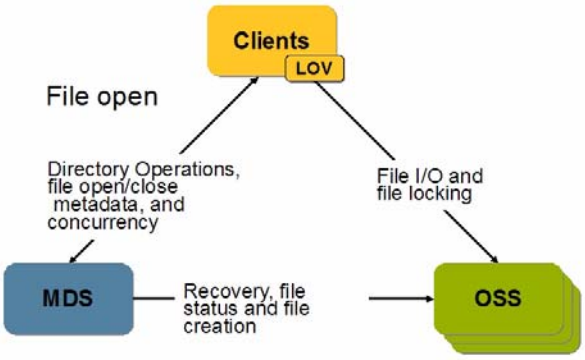
1.2.2 Management Server (MGS)

The MGS stores configuration information for all Lustre file systems in a cluster. Each Lustre target contacts the MGS to provide information, and Lustre clients contact the MGS to retrieve information. The MGS requires its own disk for storage. However, there is a provision that allows the MGS to share a disk ("co-locate") with a single MDT. The MGS is not considered "part" of an individual file system; it provides configuration information for all managed Lustre file systems to other Lustre components.

1.3 Lustre Systems

Lustre components work together as coordinated systems to manage file and directory operations in the file system (see FIGURE 1-2).

FIGURE 1-2 Lustre system interaction in a file system

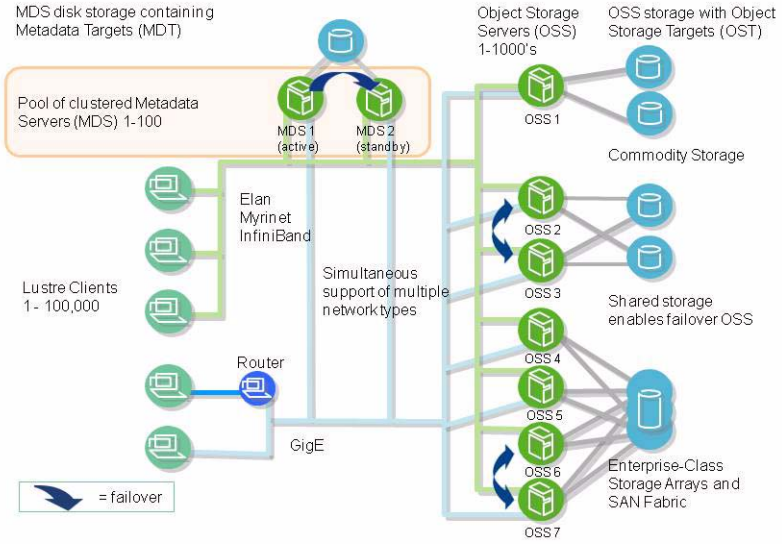


The characteristics of the Lustre system include:

	Typical number of systems	Performance	Required attached storage	Desirable hardware characteristics
Clients	1-100,000	1 GB/sec I/O, 1,000 metadata ops/sec	None	None
OSS	1-1,000	500-2.5 GB/sec	File system capacity/OSS count	Good bus bandwidth
MDS	2 (2-100 in future)	3,000-15,000 metadata ops/sec	1-2% of file system capacity	Adequate CPU power, plenty of memory

At scale, the Lustre cluster can include up to 1,000 OSSs and 100,000 clients (see FIGURE 1-3).

FIGURE 1-3 Lustre cluster at scale



1.4 Files in the Lustre File System

Traditional UNIX disk file systems use inodes, which contain lists of block numbers where file data for the inode is stored. Similarly, for each file in a Lustre file system, one inode exists on the MDT. However, in Lustre, the inode on the MDT does not point to data blocks, but instead, points to one or more objects associated with the files. This is illustrated in FIGURE 1-4. These objects are implemented as files on the OST file systems and contain file data.

FIGURE 1-4 MDS inodes point to objects, ext3 inodes point to data

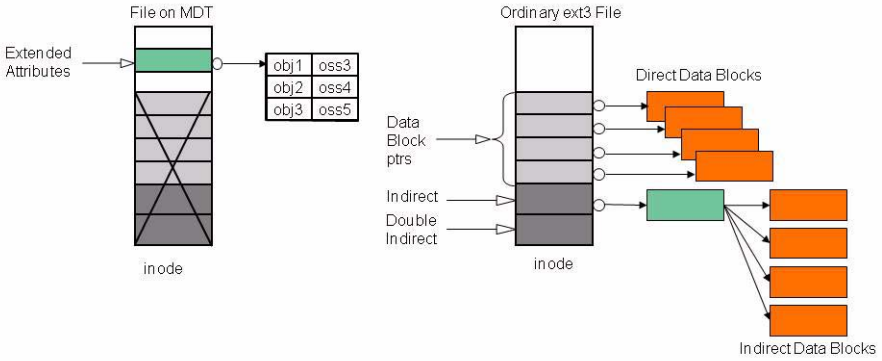
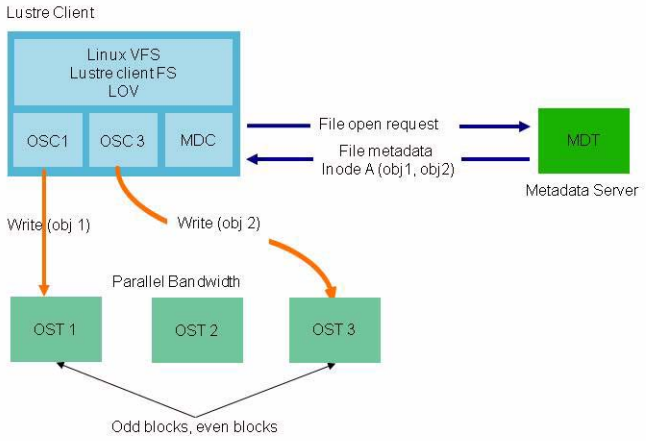


FIGURE 1-5 shows how a file open operation transfers the object pointers from the MDS to the client when a client opens the file, and how the client uses this information to perform I/O on the file, directly interacting with the OSS nodes where the objects are stored.

FIGURE 1-5 File open and file I/O in Lustre



If only one object is associated with an MDS inode, that object contains all of the data in that Lustre file. When more than one object is associated with a file, data in the file is "striped" across the objects.

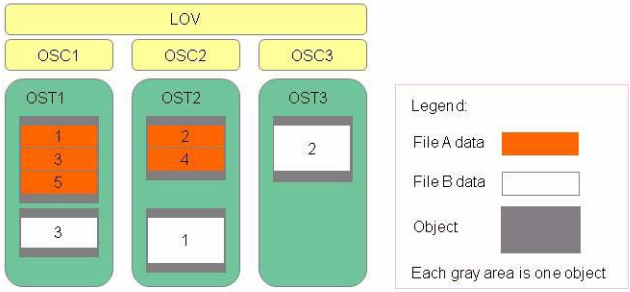
The MDS knows the layout of each file, the number and location of the file's stripes. The clients obtain the file layout from the MDS. Client do I/O against the stripes of a file by communicating directly with the relevant OSTs.

The benefits of the Lustre arrangement are clear. The capacity of a Lustre file system equals the sum of the capacities of the storage targets. The aggregate bandwidth available in the file system equals the aggregate bandwidth offered by the OSSs to the targets. Both capacity and aggregate I/O bandwidth scale simply with the number of OSSs.

1.4.1 Lustre File System and Striping

Striping allows parts of files to be stored on different OSTs, as shown in FIGURE 1-6. A RAID 0 pattern, in which data is "striped" across a certain number of objects, is used; the number of objects is called the stripe_count. Each object contains "chunks" of data. When the "chunk" being written to a particular object exceeds the stripe_size, the next "chunk" of data in the file is stored on the next target.

FIGURE 1-6 Files striped with a stripe count of 2 and 3 with different stripe sizes



File striping presents several benefits. One is that the maximum file size is not limited by the size of a single target. Lustre can stripe files over up to 160 targets, and each target can support a maximum disk use of 8 TB[3] by a file. This leads to a maximum disk use of 1.48 PB[4] by a file. Note that the maximum file size is much larger (2^64 bytes), but the file cannot have more than 1.48 PB[2] of allocated data; hence a file larger than 1.48 PB[2] must have many sparse sections. While a single file can only be striped over 160 targets, Lustre file systems have been built with almost 5000 targets, which is enough to support a 40 PB file system.

3. In Lustre 1.8.2, 16 TB on RHEL 5.
4. In Lustre 1.8.2, 2.96 PB on RHEL 5.

Another benefit of striped files is that the I/O bandwidth to a single file is the aggregate I/O bandwidth to the objects in a file and this can be as much as the bandwidth of up to 160 servers.

1.4.2 Lustre Storage

The storage attached to the servers is partitioned, optionally organized with logical volume management (LVM) and formatted as file systems. Lustre OSS and MDS servers read, write and modify data in the format imposed by these file systems.

1.4.2.1 OSS Storage

Each OSS can manage multiple object storage targets (OSTs), one for each volume; I/O traffic is load-balanced against servers and targets. An OSS should also balance network bandwidth between the system network and attached storage to prevent network bottlenecks. Depending on the server's hardware, an OSS typically serves between 2 and 25 targets, with each target up to 8 terabytes (TBs) in size.

1.4.2.2 MDS Storage

For the MDS nodes, storage must be attached for Lustre metadata, for which 1-2 percent of the file system capacity is needed. The data access pattern for MDS storage is different from the OSS storage: the former is a metadata access pattern with many seeks and read-and-writes of small amounts of data, while the latter is an I/O access pattern, which typically involves large data transfers.

High throughput to MDS storage is not important. Therefore, we recommend that a different storage type be used for the MDS (for example FC or SAS drives, which provide much lower seek times). Moreover, for low levels of I/O, RAID 5/6 patterns are not optimal, a RAID 0+1 pattern yields much better results.

Lustre uses journaling file system technology on the targets, and for a MDS, an approximately 20 percent performance gain can sometimes be obtained by placing the journal on a separate device. Typically, the MDS requires CPU power; we recommend at least four processor cores.

1.4.3 Lustre System Capacity

Lustre file system capacity is the sum of the capacities provided by the targets.

As an example, 64 OSSs, each with two 8-TB targets, provide a file system with a capacity of nearly 1 PB. If this system uses sixteen 1-TB SATA disks, it may be possible to get 50 MB/sec from each drive, providing up to 800 MB/sec of disk bandwidth. If this system is used as storage backend with a system network like InfiniBand that supports a similar bandwidth, then each OSS could provide 800 MB/sec of end-to-end I/O throughput. Note that the OSS must provide inbound and outbound bus throughput of 800 MB/sec simultaneously. The cluster could see aggregate I/O bandwidth of 64x800, or about 50 GB/sec. Although the architectural constraints described here are simple, in practice it takes careful hardware selection, benchmarking and integration to obtain such results.

In a Lustre file system, storage is only attached to server nodes, not to client nodes. If failover capability is desired, then this storage must be attached to multiple servers. In all cases, the use of storage area networks (SANs) with expensive switches can be avoided, because point-to-point connections between the servers and the storage arrays normally provide the simplest and best attachments.

1.5 Lustre Configurations

Lustre file systems are easy to configure. First, the Lustre software is installed, and then MDT and OST partitions are formatted using the standard UNIX `mkfs` command. Next, the volumes carrying the Lustre file system targets are mounted on the server nodes as local file systems. Finally, the Lustre client systems are mounted (in a manner similar to NFS mounts).

The configuration commands listed below are for the Lustre cluster shown in FIGURE 1-7.

On the MDS (*mds.your.org@tcp0*):

```
mkfs.lustre --mdt --mgs --fsname=large-fs /dev/sda
mount -t lustre /dev/sda /mnt/mdt
```

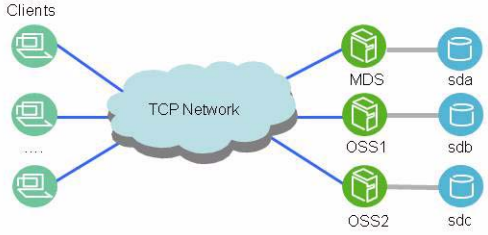
On OSS1:

```
mkfs.lustre --ost --fsname=large-fs --mgsnode=mds.your.org@tcp0 /dev/sdb
mount -t lustre /dev/sdb/mnt/ost1
```

On OSS2:

```
mkfs.lustre --ost --fsname=large-fs --mgsnode=mds.your.org@tcp0 /dev/sdc
mount -t lustre /dev/sdc/mnt/ost2
```

FIGURE 1-7 A simple Lustre cluster



1.6 Lustre Networking

In clusters with a Lustre file system, the system network connects the servers and the clients. The disk storage behind the MDSs and OSSs connects to these servers using traditional SAN technologies, but this SAN does not extend to the Lustre client system. Servers and clients communicate with one another over a custom networking API known as Lustre Networking (LNET). LNET interoperates with a variety of network transports through Network Abstraction Layers (NAL).

Key features of LNET include:

- RDMA, when supported by underlying networks such as Elan, Myrinet and InfiniBand.
- Support for many commonly-used network types such as InfiniBand and IP.
- High availability and recovery features enabling transparent recovery in conjunction with failover servers.
- Simultaneous availability of multiple network types with routing between them.

LNET includes LNDs to support many network type including:

- InfiniBand: OpenFabrics versions 1.0 and 1.2, Mellanox Gold, Cisco, Voltaire, and Silverstorm
- TCP: Any network carrying TCP traffic, including GigE, 10GigE, and IPoIB
- Quadrics: Elan3, Elan4
- Myrinet: GM, MX
- Cray: Seastar, RapidArray

The LNDs that support these networks are pluggable modules for the LNET software stack.

LNET offers extremely high performance. It is common to see end-to-end throughput over GigE networks in excess of 110 MB/sec, InfiniBand double data rate (DDR) links reach bandwidths up to 1.5 GB/sec, and 10GigE interfaces provide end-to-end bandwidth of over 1 GB/sec.

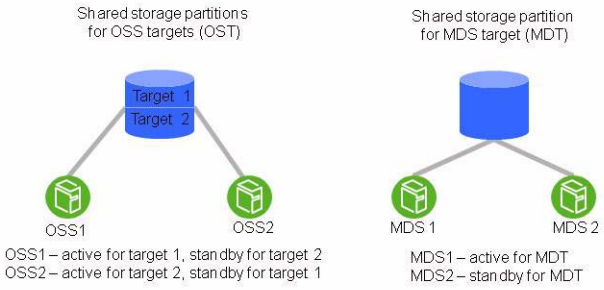
1.7 Lustre Failover and Rolling Upgrades

Lustre offers a robust, application-transparent failover mechanism that delivers call completion. This failover mechanism, in conjunction with software that offers interoperability between versions, is used to support rolling upgrades of file system software on active clusters.

The Lustre recovery feature allows servers to be upgraded without taking down the system. The server is simply taken offline, upgraded and restarted (or failed over to a standby server with the new software). All active jobs continue to run without failures, they merely experience a delay.

Lustre MDSs are configured as an active/passive pair, while OSSs are typically deployed in an active/active configuration that provides redundancy without extra overhead, as shown in FIGURE 1-8. Often the standby MDS is the active MDS for another Lustre file system, so no nodes are idle in the cluster.

FIGURE 1-8 Lustre failover configurations for OSSs and MDSs



Although a file system checking tool (lfsck) is provided for disaster recovery, journaling and sophisticated protocols re-synchronize the cluster within seconds, without the need for a lengthy fsck. Lustre version interoperability between successive minor versions is guaranteed. As a result, the Lustre failover capability is used regularly to upgrade the software without cluster downtime.

Note – Lustre does not provide redundancy for data; it depends exclusively on redundancy of backing storage devices. The backing OST storage should be RAID 5 or, preferably, RAID 6 storage. MDT storage should be RAID 1 or RAID 0+1.

CHAPTER 2

Understanding Lustre Networking

This chapter describes Lustre Networking (LNET) and supported networks, and includes the following sections:

2.1 Introduction to LNET

In a Lustre network, servers and clients communicate with one another using LNET, a custom networking API which abstracts away all transport-specific interaction. In turn, LNET operates with a variety of network transports through Lustre Network Drivers (LNDs).

The following terms are important to understanding LNET.

- **LND**: Lustre Network Driver. A modular sub-component of LNET that implements one of the network types. LNDs are implemented as individual kernel modules (or a library in userspace) and, typically, must be compiled against the network driver software.
- **Network**: A group of nodes that communicate directly with each other. The network is how LNET represents a single cluster. Multiple networks can be used to connect clusters together. Each network has a unique type and number (for example, tcp0, tcp1, or elan0).
- **NID**: Lustre Network Identifier. The NID uniquely identifies a Lustre network endpoint, including the node and the network type. There is an NID for every network which a node uses.

Key features of LNET include:

- RDMA, when supported by underlying networks such as Elan, Myrinet, and InfiniBand
- Support for many commonly-used network types such as InfiniBand and TCP/IP
- High availability and recovery features enabling transparent recovery in conjunction with failover servers
- Simultaneous availability of multiple network types with routing between them

LNET is designed for complex topologies, superior routing capabilities and simplified configuration.

2.2 Supported Network Types

LNET supports the following network types:

- TCP
- openib (Mellanox-Gold InfiniBand)
- cib (Cisco Topspin)
- iib (Infinicon InfiniBand)
- vib (Voltaire InfiniBand)
- o2ib (OFED - InfiniBand and iWARP)
- ra (RapidArray)
- Elan (Quadrics Elan)
- GM and MX (Myrinet)
- Cray Seastar

2.3 Designing Your Lustre Network

Before you configure Lustre, it is essential to have a clear understanding of the Lustre network topologies.

2.3.1 Identify All Lustre Networks

A network is a group of nodes that communicate directly with one another. As previously mentioned in this manual, Lustre supports a variety of network types and hardware, including TCP/IP, Elan, varieties of InfiniBand, Myrinet and others. The normal rules for specifying networks apply to Lustre networks. For example, two TCP networks on two different subnets (tcp0 and tcp1) would be considered two different Lustre networks.

2.3.2 Identify Nodes to Route Between Networks

Any node with appropriate interfaces can route LNET between different networks—the node may be a server, a client, or a standalone router. LNET can route across different network types (such as TCP-to-Elan) or across different topologies (such as bridging two InfiniBand or TCP/IP networks).

2.3.3 Identify Network Interfaces to Include/Exclude from LNET

If not explicitly specified, LNET uses either the first available interface or a pre-defined default for a given network type. If there are interfaces that LNET should not use (such as administrative networks, IP over IB, and so on), then the included interfaces should be explicitly listed.

2.3.4 Determine Cluster-wide Module Configuration

The LNET configuration is managed via module options, typically specified in /etc/modprobe.conf or /etc/modprobe.conf.local (depending on the distribution). To ease the maintenance of large clusters, you can configure the networking setup for all nodes using a single, unified set of options in the modprobe.conf file on each node. For more information, see the ip2nets option in Setting Up modprobe.conf for Load Balancing.

Users of liblustre should set the accept=all parameter. For details, see Module Parameters.

2.3.5 Determine Appropriate Mount Parameters for Clients

In mount commands, clients use the NID of the MDS host to retrieve their configuration information. Since an MDS may have more than one NID, a client should use the appropriate NID for its local network. If you are unsure which NID to use, there is a lctl command that can help.

MDS

On the MDS, run:

```
lctl list_nids
```

This displays the server's NIDs (networks configured to work with Lustre).

Client

On a client, run:

```
lctl which_nid <NID list>
```

This displays the closest NID for the client.

Client with SSH Access

From a client with SSH access to the MDS, run:

```
mds_nids=`ssh the_mds lctl list_nids`
lctl which_nid $mds_nids
```

This displays, generally, the correct NID to use for the MDS in the `mount` command.

Note – In the `mds_nids` command above, be sure to use the correct mark (`), not a straight quotation mark (').

2.4 Configuring LNET

This section describes how to configure LNET, including entries in the modprobe.conf file which tell LNET which NIC(s) will be configured to work with Lustre, and parameters that specify the routing that will be used with Lustre.

Note – We recommend that you use dotted-quad IP addressing rather than host names. We have found this aids in reading debug logs, and helps greatly when debugging configurations with multiple interfaces.

2.4.1 Module Parameters

LNET hardware and routing are configured via module parameters of the LNET and LND-specific modules. Parameters should be specified in the /etc/modprobe.conf or /etc/modules.conf file. This example specifies that the node should use a TCP interface and an Elan interface:

```
options lnet networks=tcp0,elan0
```

Depending on the LNDs used, it may be necessary to specify explicit interfaces. For example, if you want to use two TCP interfaces (tcp0 and tcp1, for example), it is necessary to specify the module parameters and eth*X* interfaces, like this:

```
options lnet networks=tcp0(eth0),tcp1(eth1)
```

This modprobe.conf entry specifies:

- First Lustre network, tcp0, is configured on interface eth0
- Second Lustre network, tcp1, is configured on interface eth1

Note – The requirement to specify explicit interfaces is not consistent across all LNDs used with Lustre, and LND behavior may change over time. We recommend that if your multi-homed settings do not work, try specifying the eth*X* interfaces in the `options lnet networks` line.

All LNET routers that bridge two networks are equivalent; their configuration is not primary or secondary. All available routers balance their overall load. With the router checker configured, Lustre nodes can detect router health status, avoid those that appear dead, and reuse the ones that restore service after failures. To do this, LNET routing must correspond exactly with the Linux nodes' map of alive routers. There is no hard limit on the number of LNET routers.

Note – When multiple interfaces are available during the network setup, Lustre choose the 'best' route. Once the network connection is established, Lustre expects the network to stay connected. In a Lustre network, connections do not fail over to the other interface, even if multiple interfaces are available on the same node.

Under Linux 2.6, the LNET configuration parameters can be viewed under /sys/module/; generic and acceptor parameters under `lnet` and LND-specific parameters under the corresponding LND name.

Note – Depending on the Linux distribution, options with included commas may need to be escaped using single and/or double quotes. Worst-case quotes look like:

```
options lnet'networks="tcp0,elan0"' 'routes="tcp [2,10]@elan0"'
```

Additional quotes may confuse some distributions. Check for messages such as:

```
lnet: Unknown parameter ''networks'
```

After modprobe LNET, remove the additional single quotes (modprobe.conf in this case). Additionally, the refusing connection - no matching NID message generally points to an error in the LNET module configuration.

Note – By default, Lustre ignores the loopback (lo0) interface. Lustre does not ignore IP addresses aliased to the loopback. In this case, specify all Lustre networks.

The liblustre network parameters may be set by exporting the environment variables LNET_NETWORKS, LNET_IP2NETS and LNET_ROUTES. Each of these variables uses the same parameters as the corresponding modprobe option.

Note, it is very important that a liblustre client includes ALL the routers in its setting of LNET_ROUTES. A liblustre client cannot accept connections, it can only create connections. If a server sends remote procedure call (RPC) replies via a router to which the liblustre client has not already connected, then these RPC replies are lost.

Note – Liblustre is not required or even recommended for running Lustre on Linux. Most users will not use liblustre. Instead, you should use the Lustre (VFS) client file system to mount Lustre directly. Liblustre does NOT support multi-threaded applications.

Note – Liblustre is not widely tested as part of Lustre release testing, and is currently maintained only as a courtesy to the Lustre community.

2.4.1.1 Using Usocklnd

Lustre now offers usocklnd, a socket-based LND that uses TCP in userspace. By default, liblustre is compiled with usocklnd as the transport, so there is no need to specially enable it.

Use the following environmental variables to tune usocklnd's behavior.

Variable	Description
USOCK_SOCKNAGLE=N	Turns the TCP Nagle algorithm on or off. Setting N to 0 (the default value), turns the algorithm off. Setting N to 1 turns the algorithm on.
USOCK_SOCKBUFSIZ=N	Changes the socket buffer size. Setting N to 0 (the default value), specifies the default socket buffer size. Setting N to another value (must be a positive integer) causes usocklnd to try to set the socket buffer size to the specified value.
USOCK_TXCREDITS=N	Specifies the maximum number of concurrent sends. The default value is 256. N should be set to a positive value.
USOCK_PEERTXCREDITS=N	Specifies the maximum number of concurrent sends per peer. The default value is 8. N should be set to a positive value and should not be greater than the value of the USOCK_TXCREDITS parameter.
USOCK_NPOLLTHREADS=N	Defines the degree of parallelism of usocklnd, by equaling the number of threads devoted to processing network events. The default value is the number of CPUs in the system. N should be set to a positive value.

USOCK_FAIR_LIMIT=N	The maximum number of times that usocklnd loops processing events before the next polling occurs. The default value is 1, meaning that every network event has only one chance to be processed before polling occurs the next time. N should be set to a positive value.
USOCK_TIMEOUT=N	Specifies the network timeout (measured in seconds). Network options that are not completed in N seconds time out and are canceled. The default value is 50 seconds. N should be a positive value.
USOCK_POLL_TIMEOUT=N	Specifies the polling timeout; how long usocklnd 'sleeps' if no network events occur. N results in a slightly lower overhead of checking network timeouts and longer delay of evicting timed-out events. The default value is 1 second. N should be set to a positive value.
USOCK_MIN_BULK=N	This tunable is only used for typed network connections. Currently, liblustre clients do not use this usocklnd facility.

2.4.1.2 OFED InfiniBand Options

For the SilverStorm/Infinicon InfiniBand LND (iiblnd), the network and HCA may be specified, as in this example:

```
options lnet networks="o2ib3(ib3)"
```

This specifies that the node is on o2ib network number 3, using HCA ib3.

2.4.2 Module Parameters - Routing

The following parameter specifies a colon-separated list of router definitions. Each route is defined as a network number, followed by a list of routers.

```
route=<net type> <router NID(s)>
```

Examples:

```
options lnet 'networks="o2ib0"' 'routes="tcp0 192.168.10.[1-8]@o2ib0"'
```

This is an example for IB clients to access TCP servers via 8 IB-TCP routers.

```
options lnet 'ip2nets="tcp0 10.10.0.*; o2ib0(ib0) 192.168.10.[1-128]"' \
'routes="tcp 192.168.10.[1-8]@o2ib0; o2ib 10.10.0.[1-8]@tcp0"
```

This specifies bi-directional routing; TCP clients can reach Lustre resources on the IB networks and IB servers can access the TCP networks. For more information on ip2nets, Modprobe.conf.

Note – Configure IB network interfaces on a different subnet than LAN interfaces.

Best Practices for ip2nets, routes and networks Options

For the `ip2nets`, `routes` and `networks` options, several best practices must be followed or configuration errors occur.

Best Practice 1: If you add a comment to any of the above options, position the semicolon after the comment. If you fail to do so, some nodes are not properly initialized because LNET silently ignores everything following the '#' character (which begins the comment), until it reaches the next semicolon. This is subtle; no error message is generated to alert you to the problem.

This example shows the correct syntax:

```
options lnet ip2nets="pt10 192.168.0.[89,93] # comment with semicolon AFTER comment; \
                      pt11 192.168.0.[92,96] # comment
```

In this example, the following is ignored: `comment with semicolon AFTER comment`

This example shows the wrong syntax:

```
options lnet ip2nets="pt10 192.168.0.[89,93]; # comment with semicolon BEFORE comment \
                      pt11 192.168.0.[92,96];
```

In this example, the following is ignored: `comment with semicolon BEFORE comment pt11 192.168.0.[92,96]`. Because LNET silently ignores `pt11 192.168.0.[92,96]`, these nodes are not properly initialized.

Best Practice 2: Do not add an excessive number of comments to these options. The Linux kernel has a limit on the length of string module options; it is usually 1KB, but may differ in vendor kernels. If you exceed this limit, errors result and the configuration specified by the user is not processed properly.

Using Routing Parameters Across a Cluster

To ease Lustre administration, the same routing parameters can be used across different parts of a routed cluster. For example, the bi-directional routing example above can be used on an entire cluster (TCP clients, TCP-IB routers, and IB servers):

- TCP clients would ignore o2ib0(ib0) 192.168.10.[1-128] in ip2nets since they have no such interfaces. Similarly, IB servers would ignore tcp0 192.168.0.*. But TCP-IB routers would use both since they are multi-homed.
- TCP clients would ignore the route "tcp 192.168.10.[1-8]@o2ib0" since the target network is a local network. For the same reason, IB servers would ignore "o2ib 10.10.0.[1-8]@tcp0".

- TCP-IB routers would ignore both routes, because they are multi-homed. Moreover, the routers would enable LNet forwarding since their NIDs are specified in the 'routes' parameters as being routers.

live_router_check_interval, dead_router_check_interval, auto_down, check_routers_before_use and router_ping_timeout

In a routed Lustre setup with nodes on different networks such as TCP/IP and Elan, the router checker checks the status of a router. The `auto_down` parameter enables/disables (1/0) the automatic marking of router state.

The `live_router_check_interval` parameter specifies a time interval in seconds after which the router checker will ping the live routers.

In the same way, you can set the `dead_router_check_interval` parameter for checking dead routers.

You can set the timeout for the router checker to check the live or dead routers by setting the `router_ping_timeout` parameter. The Router pinger sends a ping message to a dead/live router once every `dead/live_router_check_interval` seconds, and if it does not get a reply message from the router within `router_ping_timeout` seconds, it considers the router to be down.

The last parameter is `check_routers_before_use`, which is off by default. If it is turned on, you must also give `dead_router_check_interval` a positive integer value.

The router checker gets the following variables for each router:

- Last time that it was disabled
- Duration of time for which it is disabled

The initial time to disable a router should be one minute (enough to plug in a cable after removing it). If the router is administratively marked as "up", then the router checker clears the timeout. When a route is disabled (and possibly new), the "sent packets" counter is set to 0. When the route is first re-used (that is an elapsed disable time is found), the sent packets counter is incremented to 1, and incremented for all further uses of the route. If the route has been used for 100 packets successfully, then the sent-packets counter should be with a value of 100. Set the timeout to 0 (zero), so future errors no longer double the timeout.

Note – The `router_ping_timeout` is consistent with the default LND timeouts. You may have to increase it on very large clusters if the LND timeout is also increased. For larger clusters, we suggest increasing the check interval.

2.4.2.1 LNET Routers

All LNET routers that bridge two networks are equivalent. They are not configured as primary or secondary, and load is balanced across all available routers.

With the router checker configured, Lustre nodes can detect router health status, avoid those that appear dead, and reuse the ones that restore service after failures.

There are no hard requirements regarding the number of LNET routers, although there should enough to handle the required file serving bandwidth (and a 25% margin for headroom).

Comparing 32-bit and 64-bit LNET Routers

By default, at startup, LNET routers allocate 544M (i.e. 139264 4K pages) of memory as router buffers. The buffers can only come from low system memory (i.e. ZONE_DMA and ZONE_NORMAL).

On 32-bit systems, low system memory is, at most, 896M no matter how much RAM is installed. The size of the default router buffer puts big pressure on low memory zones, making it more likely that an out-of-memory (OOM) situation will occur. This is a known cause of router hangs. Lowering the value of the large_router_buffers parameter can circumvent this problem, but at the cost of penalizing router performance, by making large messages wait for longer for buffers.

On 64-bit architectures, the ZONE_HIGHMEM zone is always empty. Router buffers can come from all available memory and out-of-memory hangs do not occur. Therefore, we recommend using 64-bit routers.

2.4.3 Downed Routers

There are two mechanisms to update the health status of a peer or a router:

- LNET can actively check health status of all routers and mark them as dead or alive automatically. By default, this is off. To enable it set `auto_down` and if desired `check_routers_before_use`. This initial check may cause a pause equal to `router_ping_timeout` at system startup, if there are dead routers in the system.
- When there is a communication error, all LNDs notify LNET that the peer (not necessarily a router) is down. This mechanism is always on, and there is no parameter to turn it off. However, if you set the LNET module parameter `auto_down` to 0, LNET ignores all such peer-down notifications.

Several key differences in both mechanisms:

- The router pinger only checks routers for their health, while LNDs notices all dead peers, regardless of whether they are a router or not.
- The router pinger actively checks the router health by sending pings, but LNDs only notice a dead peer when there is network traffic going on.
- The router pinger can bring a router from alive to dead or vice versa, but LNDs can only bring a peer down.

2.5 Starting and Stopping LNET

Lustre automatically starts and stops LNET, but it can also be manually started in a standalone manner. This is particularly useful to verify that your networking setup is working correctly before you attempt to start Lustre.

2.5.1 Starting LNET

To start LNET, run:

```
$ modprobe lnet
$ lctl network up
```

To see the list of local NIDs, run:

```
$ lctl list_nids
```

This command tells you the network(s) configured to work with Lustre

If the networks are not correctly setup, see the modules.conf "networks=" line and make sure the network layer modules are correctly installed and configured.

To get the best remote NID, run:

```
$ lctl which_nid <NID list>
```

where `<NID list>` is the list of available NIDs.

This command takes the "best" NID from a list of the NIDs of a remote host. The "best" NID is the one that the local node uses when trying to communicate with the remote node.

2.5.1.1 Starting Clients

To start a TCP client, run:

```
mount -t lustre mdsnode:/mdsA/client /mnt/lustre/
```

To start an Elan client, run:

```
mount -t lustre 2@elan0:/mdsA/client /mnt/lustre
```

2.5.2 Stopping LNET

Before the LNET modules can be removed, LNET references must be removed. In general, these references are removed automatically when Lustre is shut down, but for standalone routers, an explicit step is needed to stop LNET. Run:

```
lctl network unconfigure
```

Note – Attempting to remove Lustre modules prior to stopping the network may result in a crash or an LNET hang. if this occurs, the node must be rebooted (in most cases). Make sure that the Lustre network and Lustre are stopped prior to unloading the modules. Be extremely careful using `rmmod -f`.

To unconfigure the LNET network, run:

```
modprobe -r <any lnd and the lnet modules>
```

Tip – To remove all Lustre modules, run:

```
$ lctl modules | awk '{print $2}' | xargs rmmod
```

PART II Lustre Administration

Lustre administration includes the steps necessary to meet pre-installation requirements, and install and configure Lustre. It also includes advanced topics such as failover, quotas, bonding, benchmarking, Kerberos and POSIX.

CHAPTER 3

Installing Lustre

Lustre installation involves two procedures, meeting the installation prerequisites and installing the Lustre software, either from RPMs or from source code. This chapter includes these sections:

- Preparing to Install Lustre
- Installing Lustre from RPMs
- Installing Lustre from Source Code

Lustre can be installed from either packaged binaries (RPMs) or freely-available source code. Installing from the package release is straightforward, and recommended for new users. Integrating Lustre into an existing kernel and building the associated Lustre software is an involved process.

For either installation method, the following are required:

- Linux kernel patched with Lustre-specific patches
- Lustre modules compiled for the Linux kernel
- Lustre utilities required for Lustre configuration

Note – When installing Lustre and creating components on devices, a certain amount of space is reserved, so less than 100% of storage space will be available. Lustre servers use the ext3 file system to store user-data objects and system data. By default, ext3 file systems reserve 5% of space that cannot be used by Lustre. Additionally, Lustre reserves up to 400 MB on each OST for journal use[1]. This reserved space is unusable for general storage. For this reason, you will see up to 400 MB of space used on each OST before any file object data is saved to it.

1. Additionally, a few bytes outside the journal are used to create accounting data for Lustre.

3.1 Preparing to Install Lustre

To successfully install and run Lustre, make sure the following installation prerequisites have been met:

- Supported Operating System, Platform and Interconnect
- Required Lustre Software
- Required Tools and Utilities
- (Optional) High-Availability Software
- Debugging Tools
- Environmental Requirements
- Memory Requirements

3.1.1 Supported Operating System, Platform and Interconnect

Lustre 1.8 supports the following operating systems, platforms[2] and interconnects. To install Lustre from downloaded packages (RPMs), you must use a supported configuration.

Configuration Component	Supported Type
Operating system	OEL 5.4, i686 and x86_64 only (Lustre 1.8.2) OEL 5.3, i686 and x86_64 only (Lustre 1.8.1.1 and 1.8.2) Red Hat Enterprise Linux 5 SuSE Linux Enterprise Server 10 SuSE Linux Enterprise Server 11, i686 and x86_64 only (Lustre 1.8.1 and later) Linux kernel 2.6.16 or greater **NOTE**: Lustre does not support security-enhanced (SE) Linux (including clients and servers).
Platform	x86, IA-64, x86-64 (EM64 and AMD64) PowerPC architectures (for clients only) and mixed-endian clusters
Interconnect	TCP/IP Quadrics Elan 3 and 4 Myri-10G and Myrinet-2000 Mellanox InfiniBand (Voltaire, OpenIB, Silverstorm and any OFED-supported InfiniBand adapter)

Note – Lustre clients running on architectures with different endianness are supported. One limitation is that the PAGE_SIZE kernel macro on the client must be as large as the PAGE_SIZE of the server. In particular, ia64 clients with large pages (up to 64kB pages) can run with i386 servers (4kB pages). If you are running i386 clients with ia64 servers, you must compile the ia64 kernel with a 4kB PAGE_SIZE (so the server page size is not larger than the client page size).

2. We encourage the use of 64-bit platforms.

3.1.2 Required Lustre Software

To install Lustre, the following are required:

- Linux kernel patched with Lustre-specific patches (the patched Linux kernel is required only on the Lustre MDS and OSSs)
- Lustre modules compiled for the Linux kernel
- Lustre utilities required for Lustre configuration
- (Optional) Network-specific kernel modules and libraries (for example, kernel modules and libraries required for an InfiniBand interconnect)

These packages can be downloaded from the Lustre download site.

3.1.3 Required Tools and Utilities

Several third-party utilities are required:

- **e2fsprogs**: Lustre requires a recent version of e2fsprogs that understands extents. Use `e2fsprogs-1.41-6` or later, available on the Lustre download site.

Note – Lustre-patched e2fsprogs utility only needs to be installed on machines that mount backend (ldiskfs) file systems, such as the OSS, MDS and MGS nodes. It does not need to be loaded on clients.

- **Perl** - Various userspace utilities are written in Perl. Any recent version of Perl will work with Lustre.

3.1.4 (Optional) High-Availability Software

If you plan to enable failover server functionality with Lustre (either on an OSS or the MDS), you must add high-availability (HA) software to your cluster software. You can use any HA software package with Lustre.[3] For more information, see Failover.

3. In this manual, the Linux-HA (Heartbeat) package is referenced, but you can use any HA software.

3.1.5 Debugging Tools

Lustre is a complex system and you may encounter problems when using it. You should have debugging tools on hand to help figure out how and why a problem occurred. The e2fsprogs package (available on the Lustre download site), includes the Lustre debugfs tool, which can be can used to interactively debug an ext3/ldiskfs[4] file system. The debugfs utility can either be used either to check status of or modify information in the file system.

There are also several third-party tools you can use, such as GDB, coupled with crash. These tools can be used to investigate live systems and kernel core dumps. There are also useful kernel patches/ modules, such as netconsole and netdump, that allow core dumps to be made across the network.

For more information, see these websites:

Third-party Tool	URL
GDB	http://www.gnu.org/software/gdb/gdb.html
crash	http://oss.missioncriticallinux.com/projects/crash/
netconsole	http://lwn.net/2001/0927/a/netconsole.php3
netdump	http://www.redhat.com/support/wpapers/redhat/netdump/

4. ldiskfs is the Sun development version of ext4.

3.1.6 Environmental Requirements

Make sure the following environmental requirements are met before installing Lustre:

- (Recommended) **Provide remote shell access to clients**. Although not strictly required to run Lustre, we recommend that all cluster nodes have remote shell client access, to facilitate the use of Lustre configuration and monitoring scripts. Parallel Distributed SHell (pdsh) is preferable, although Secure SHell (SSH) is acceptable.
- **Ensure client clocks are synchronized**. Lustre uses client clocks for timestamps. If clocks are out-of-sync between clients and servers, timeouts and client evictions will occur. Drifting clocks can also cause problems by, for example, making it difficult to debug multi-node issues or correlate logs, which depend on timestamps. We recommend that you use Network Time Protocol (NTP) to keep client and server clocks in sync with each other. For more information about NTP, see: http://www.ntp.org.
- **Maintain uniform file access permissions on all cluster nodes**. Use the same user IDs (UID) and group IDs (GID) on all clients. If use of supplemental groups is required, verify that the group_upcall requirements have been met. See User/Group Cache Upcall.
- (Recommended) **Disable Security-Enhanced Linux (SELinux) on servers and clients**. Lustre does not support SELinux. Therefore, disable the SELinux system extension on all Lustre nodes and make sure other security extensions, like Novell AppArmorand network packet filtering tools (such as iptables) do not interfere with Lustre.

3.1.7 Memory Requirements

This section describes the memory requirements of Lustre.

3.1.7.1 MDS Memory Requirements

MDS memory requirements are determined by the following factors:

- Number of clients
- Size of the directories
- Extent of load

The amount of memory used by the MDS is a function of how many clients are on the system, and how many files they are using in their working set. This is driven, primarily, by the number of locks a client can hold at one time. The default maximum number of locks for a compute node is 100*num_cores, and interactive clients can hold in excess of 10,000 locks at times. For the MDS, this works out to approximately 2 KB per file, including the Lustre DLM lock and kernel data structures for it, just for the current working set.

There is, by default, 400 MB for the file system journal, and additional RAM usage for caching file data for the larger working set that is not actively in use by clients, but should be kept "HOT" for improved access times. Having file data in cache can improve metadata performance by a factor of 10x or more compared to reading it from disk. Approximately 1.5 KB/file is needed to keep a file in cache.

For example, for a single MDT on an MDS with 1,000 clients, 16 interactive nodes, and a 2 million file working set (of which 400,000 files are cached on the clients):

File system journal	= 400 MB
1000 * 4-core clients * 100 files/core * 2kB	= 800 MB
16 interactive clients * 10,000 files * 2kB	= 320 MB
1,600,000 file extra working set * 1.5kB/file	= 2400 MB

Thus, the minimum requirement for a system with this configuration is 4-GB RAM. However, additional memory may significantly improve performance[5].

If there are directories containing 1 million or more files, you may benefit significantly from having more memory. For example, in an environment where clients randomly access one of 10 million files, having extra memory for the cache significantly improves performance.

5. Having more RAM is always prudent, given the relatively low cost of this component compared to the total system cost.

3.1.7.2 OSS Memory Requirements

When planning the hardware for an OSS node, consider the memory usage of several components in the Lustre system (i.e., journal, service threads, file system metadata, etc.). Also, consider the effect of the OSS read cache feature (new in Lustre 1.8), which consumes memory as it caches data on the OSS node.

- **Journal size**: By default, each Lustre ldiskfs file system has 400 MB for the journal size. This can pin up to an equal amount of RAM on the OSS node per file system.
- **Service threads**: The service threads on the OSS node pre-allocate a 1 MB I/O buffer for each ost_io service thread, so these buffers do not need to be allocated and freed for each I/O request.
- **File system metadata**: A reasonable amount of RAM needs to be available for file system metadata. While no hard limit can be placed on the amount of file system metadata, if more RAM is available, then the disk I/O is needed less often to retrieve the metadata.
- **Network transport**: If you are using TCP or other network transport that uses system memory for send/receive buffers, this must also be taken into consideration.
- **Failover configuration**: If the OSS node will be used for failover from another node, then the RAM for each journal should be doubled, so the backup server can handle the additional load if the primary server fails.
- **OSS read cache**: OSS read cache provides read-only caching of data on an OSS, using the regular Linux page cache to store the data. Just like caching from a regular file system in Linux, OSS read cache uses as much physical memory as is available.

Because of these memory requirements, the following calculations should be taken as determining the absolute minimum RAM required in an OSS node.

Calculating OSS Memory Requirements

The minimum recommended RAM size for an OSS with two OSTs is computed below:

1.5 MB per OST IO thread * 512 threads = 768 MB

e1000 RX descriptors, RxDescriptors=4096 for 9000 byte MTU = 128 MB

Operating system overhead = 512 MB

400 MB journal size * 2 OST devices = 800 MB

600 MB file system metadata cache * 2 OSTs = 1200 MB

This consumes about 1,700 MB just for the pre-allocated buffers, and an additional 2 GB for minimal file system and kernel usage. Therefore, for a non-failover configuration, the minimum RAM would be 4 GB for an OSS node with two OSTs. While it is not strictly required, adding additional memory on the OSS will improve the performance of reading smaller, frequently-accessed files.

For a failover configuration, the minimum RAM would be at least 6 GB. For 4 OSTs on each OSS in a failover configuration 10GB of RAM is reasonable. When the OSS is not handling any failed-over OSTs the extra RAM will be used as a read cache.

As a reasonable rule of thumb, about 2 GB of base memory plus 1 GB per OST can be used. In failover configurations, about 2 GB per OST is needed.

3.2 Installing Lustre from RPMs

This procedure describes how to install Lustre from the RPM packages. This is the easier installation method and is recommended for new users.

Alternately, you can install Lustre directly from the source code. For more information on this installation method, see Installing Lustre from Source Code.

Note – In all Lustre installations, the server kernel that runs on an MDS, MGS or OSS must be patched. However, running a patched kernel on a Lustre client is optional and only required if the client will be used for multiple purposes, such as running as both a client and an OST.

Caution – Lustre contains kernel modifications which interact with storage devices and may introduce security issues and data loss if not installed, configured or administered properly. Before installing Lustre, be cautious and back up ALL data.

Use this procedure to install Lustre from RPMs.

1. **Verify that all Lustre installation requirements have been met.**

   For more information on these prerequisites, see Preparing to Install Lustre.

2. **Download the Lustre RPMs.**

   a. **On the Lustre download site, select your platform.**

      The files required to install Lustre (kernels, modules and utilities RPMs) are listed for the selected platform.

   b. **Download the required files.**

      Use the Download Manager or download the files individually.

3. **Install the Lustre packages.**

Some Lustre packages are installed on servers (MDS and OSSs), and others are installed on Lustre clients. Lustre packages must be installed in a specific order.

Caution – For a non-production Lustre environment or for testing, a Lustre client and server can run on the same machine. However, *for best performance in a production environment, dedicated clients are always best*. Performance and other issues can occur when an MDS or OSS and a client are running on the same machine[6]. The MDS and MGS can run on the same machine.

---

a. **For each Lustre package, determine if it needs to be installed on servers and/or clients. Use** TABLE 3-1 **to determine where to install a specific package. Depending on your platform, not all of the listed files need to be installed.**

TABLE 3-1 Lustre required packages, descriptions and installation guidance

Lustre Package	Description	Install on servers	Install on patchless clients	Install on patched clients
Lustre kernel RPMs				
kernel-lustre-<ver>	Lustre-patched kernel package for RHEL 5 (i686, ia64 and x86_64) platform.	X		X*
kernel-lustre-smp-<ver>	Lustre-patched kernel package for SuSE Server 10 (x86_64) platform.	X		X*
kernel-lustre-bigsmp-<ver>	Lustre-patched kernel package for SuSE Server 10 (i686) platform.	X		X*
kernel-ib-<ver>	Lustre OFED package. Install if the network interconnect is InfiniBand.	X	X	X*
kernel-lustre-default-<ver> kernel-lustre-default-base-<ver>	Lustre-patched kernel package for SuSE Server 11 (i686 and x86_64) platform.	X		X*
Lustre module RPMs				

6. Running the MDS and a client on the same machine can cause recovery and deadlock issues, and the performance of other Lustre clients to suffer. Running the OSS and a client on the same machine can cause issues with low memory and memory pressure. The client consume all of the memory and tries to flush pages to disk. The OSS needs to allocate pages to receive data from the client, but cannot perform this operation, due to low memory. This can result in OOM kill and other issues.

TABLE 3-1 Lustre required packages, descriptions and installation guidance

Lustre Package	Description	Install on servers	Install on patchless clients	Install on patched clients
lustre-modules-<ver>	Lustre modules for the patched kernel.	X		X*
lustre-client-modules-<ver>	Lustre modules for patchless clients.		X	
Lustre utilities				
lustre-<ver>	Lustre utilities package. This includes userspace utilities to configure and run Lustre.	X		X*
lustre-ldiskfs-<ver>	Lustre-patched backing file system kernel module package for the ext3 file system	X		
e2fsprogs-<ver>	Utilities package used to maintain the ext3 backing file system.	X		
lustre-client-<ver>	Lustre utilities for patchless clients		X	

* Only install this kernel RPM if you want to patch the client kernel. You do not have to patch the clients to run Lustre.

b. Install the kernel, modules and ldiskfs packages.

Use the `rpm -ivh` command to install the kernel, module and ldiskfs packages. For example:

```
$ rpm -ivh kernel-lustre-smp-<ver> \
kernel-ib-<ver> \
lustre-modules-<ver> \
lustre-ldiskfs-<ver>
```

c. Install the utilities/userspace packages.

Use the `rpm -ivh` command to install the utilities packages. For example:

```
$ rpm -ivh lustre-<ver>
```

d. Install the e2fsprogs package.

Use the `rpm -ivh` command to install the e2fsprogs package. For example:

```
$ rpm -ivh e2fsprogs-<ver>
```

If e2fsprogs is already installed on your Linux system, install the Lustre-specific e2fsprogs version by using `rpm -Uvh` to update the existing e2fsprogs package. For example:

```
$ rpm -Uvh e2fsprogs-<ver>
```

The rpm command options `--force` or `--nodeps` are not required to install or update the Lustre-specific e2fsprogs package. We specifically recommend that you not use these options. If errors are reported, notify Lustre Support by filing a bug.

e. (Optional) If you want to add optional packages to your Lustre file system, install them now.

Optional packages include file system creation and repair tools, debugging tools, test programs and scripts, Linux kernel and Lustre source code, and other packages. A complete list of optional packages for your platform is provided on the Lustre download site.

4. **Verify that the boot loader (grub.conf or lilo.conf) has been updated to load the patched kernel.**

5. **Reboot the patched clients and the servers.**

   a. If you applied the patched kernel to any clients, reboot them.

   Unpatched clients do not need to be rebooted.

   b. Reboot the servers.

Once all machines have rebooted, go to Configuring Lustre to configure Lustre Networking (LNET) and the Lustre file system.

3.3 Installing Lustre from Source Code

If you need to build a customized Lustre server kernel or are using a Linux kernel that has not been tested with the version of Lustre you are installing, you may need to build and install Lustre from source code. This involves several steps:

- Patching the core kernel
- Configuring the kernel to work with Lustre
- Creating Lustre and kernel RPMs from source code.

Please note that the Lustre/kernel configurations available at the Lustre download site have been extensively tested and verified with Lustre. The recommended method for installing Lustre servers is to use these pre-built binary packages (RPMs). For more information on this installation method, see Installing Lustre from RPMs.

Caution – Lustre contains kernel modifications which interact with storage devices and may introduce security issues and data loss if not installed, configured and administered correctly. Before installing Lustre, be cautious and back up ALL data.

Note – When using third-party network hardware with Lustre, the third-party modules (typically, the drivers) must be linked against the Linux kernel. The LNET modules in Lustre also need these references. To meet these requirements, a specific process must be followed to install and recompile Lustre. See Installing Lustre with a Third-Party Network Stack, for an example showing how to install Lustre 1.6.6 using the Myricom MX 1.2.7 driver. The same process can be used for other third-party network stacks.

3.3.1 Patching the Kernel

If you are using non-standard hardware, plan to apply a Lustre patch, or have another reason not to use packaged Lustre binaries, you have to apply several Lustre patches to the core kernel and run the Lustre configure script against the kernel.

3.3.1.1 Introducing the Quilt Utility

To simplify the process of applying Lustre patches to the kernel, we recommend that you use the Quilt utility.

Quilt manages a stack of patches on a single source tree. A series file lists the patch files and the order in which they are applied. Patches are applied, incrementally, on the base tree and all preceding patches. You can:

- Apply patches from the stack (`quilt push`)
- Remove patches from the stack (`quilt pop`)
- Query the contents of the series file (`quilt series`), the contents of the stack (`quilt applied`, `quilt previous`, `quilt top`), and the patches that are not applied at a particular moment (`quilt next`, `quilt unapplied`).
- Edit and refresh (`update`) patches with Quilt, as well as revert inadvertent changes, and fork or clone patches and show the diffs before and after work.

A variety of Quilt packages (RPMs, SRPMs and tarballs) are available from various sources. Use the most recent version you can find. Quilt depends on several other utilities, e.g., the `coreutils` RPM that is only available in RedHat 9. For other RedHat kernels, you have to get the required packages to successfully install Quilt. If you cannot locate a Quilt package or fulfill its dependencies, you can build Quilt from a tarball, available at the Quilt project website:

http://savannah.nongnu.org/projects/quilt

For additional information on using Quilt, including its commands, see Introduction to Quilt and the quilt(1) man page.

3.3.1.2 Get the Lustre Source and Unpatched Kernel

The Lustre Engineering Team has targeted several Linux kernels for use with Lustre servers (MDS/OSS) and provides a series of patches for each one. The Lustre patches are maintained in the kernel_patch directory bundled with the Lustre source code.

Note – Each patch series has been tailored to a specific kernel version, and may or may not apply cleanly to other versions of the kernel.

To obtain the Lustre source and unpatched kernel:

1. **Verify that all of the Lustre installation requirements have been met.**

   For more information on these prerequisites, see Preparing to Install Lustre.

2. **Download the Lustre source code. On the Lustre download site, select a version of Lustre to download and then select Source as the platform.**

3. **Download the unpatched kernel.**

   For convenience, Sun maintains an archive of unpatched kernel sources at:

   http://downloads.lustre.org/public/kernels/

4. **To save time later, download e2fsprogs now.**

   The source code for Sun's Lustre-enabled e2fsprogs distribution can be found at:

   http://downloads.lustre.org/public/tools/e2fsprogs/

3.3.1.3 Patch the Kernel

This procedure describes how to use Quilt to apply the Lustre patches to the kernel. To illustrate the steps in this procedure, a RHEL 5 kernel is patched for Lustre 1.6.5.1.

1. **Unpack the Lustre source and kernel to separate source trees.**

   a. **Unpack the Lustre source.**

      For this procedure, we assume that the resulting source tree is in `/tmp/lustre-1.6.5.1`

   b. **Unpack the kernel.**

      For this procedure, we assume that the resulting source tree (also known as the destination tree) is in `/tmp/kernels/linux-2.6.18`

2. **Select a config file for your kernel, located in the `kernel_configs` directory (`lustre/kernel_patches/kernel_config`).**

   The `kernel_config` directory contains the `.config` files, which are named to indicate the kernel and architecture with which they are associated. For example, the configuration file for the 2.6.18 kernel shipped with RHEL 5 (suitable for i686 SMP systems) is `kernel-2.6.18-2.6-rhel5-i686-smp.config`.

3. **Select the `series` file for your kernel, located in the `series` directory (`lustre/kernel_patches/series`).**

   The `series` file contains the patches that need to be applied to the kernel.

4. **Set up the necessary symlinks between the kernel patches and the Lustre source.**

   This example assumes that the Lustre source files are unpacked under `/tmp/lustre-1.6.5.1` and you have chosen the `2.6-rhel5.series` file). Run:

   ```
   $ cd /tmp/kernels/linux-2.6.18
   $ rm -f patches series
   $ ln -s /tmp/lustre-1.6.5.1/lustre/kernel_patches/series/2.6-\
   rhel5.series ./series
   $ ln -s /tmp/lustre-1.6.5.1/lustre/kernel_patches/patches .
   ```

5. **Use Quilt to apply the patches in the selected `series` file to the unpatched kernel. Run:**

   ```
   $ cd /tmp/kernels/linux-2.6.18
   $ quilt push -av
   ```

   The patched destination tree acts as a base Linux source tree for Lustre.

3.3.2 Create and Install the Lustre Packages

After patching the kernel, configure it to work with Lustre, create the Lustre packages (RPMs) and install them.

1. **Configure the patched kernel to run with Lustre. Run:**

```
$ cd <path to kernel tree>
$ cp /boot/config-`uname -r` .config
$ make oldconfig || make menuconfig
$ make include/asm
$ make include/linux/version.h
$ make SUBDIRS=scripts
$ make include/linux/utsrelease.h
```

2. **Run the Lustre configure script against the patched kernel and create the Lustre packages.**

```
$ cd <path to lustre source tree>
$ ./configure --with-linux=<path to kernel tree>
$ make rpms
```

This creates a set of .rpms in `/usr/src/redhat/RPMS/<arch>` with an appended date-stamp. The SuSE path is `/usr/src/packages`.

Note – You do not need to run the Lustre configure script against an unpatched kernel.

Example set of RPMs:

```
lustre-1.6.5.1-\
2.6.18_53.xx.xx.el5_lustre.1.6.5.1.custom_20081021.i686.rpm

lustre-debuginfo-1.6.5.1-\
2.6.18_53.xx.xx.el5_lustre.1.6.5.1.custom_20081021.i686.rpm

lustre-modules-1.6.5.1-\
2.6.18_53.xx.xxel5_lustre.1.6.5.1.custom_20081021.i686.rpm

lustre-source-1.6.5.1-\
2.6.18_53.xx.xx.el5_lustre.1.6.5.1.custom_20081021.i686.rpm
```

Note – If the steps to create the RPMs fail, contact Lustre Support by reporting a bug. See Reporting a Lustre Bug.

Note – Lustre supports several features and packages that extend the core functionality of Lustre. These features/packages can be enabled at the build time by issuing appropriate arguments to the `configure` command. For a list of supported features and packages, run `./configure –help` in the Lustre source tree. The `configs/` directory of the kernel source contains the config files matching each the kernel version. Copy one to `.config` at the root of the kernel tree.

3. **Create the kernel package. Navigate to the kernel source directory and run:**

   ```
   $ make rpm
   ```

   Example result:

   ```
   kernel-2.6.95.0.3.EL_lustre.1.6.5.1custom-1.i686.rpm
   ```

Note – Step 3 is only valid for RedHat and SuSE kernels. If you are using a stock Linux kernel, you need to get a script to create the kernel RPM.

4. **Install the Lustre packages.**

   Some Lustre packages are installed on servers (MDS and OSSs), and others are installed on Lustre clients. For guidance on where to install specific packages, see TABLE 3-1, which lists required packages and for each package, where to install it. Depending on the selected platform, not all of the packages listed in TABLE 3-1 need to be installed.

Note – Running the patched server kernel on the clients is optional. It is not necessary unless the clients will be used for multiple purposes, for example, to run as a client and an OST.

Lustre packages should be installed in this order:

a. **Install the kernel, modules and ldiskfs packages.**

   Navigate to the directory where the RPMs are stored, and use the `rpm -ivh` command to install the `kernel`, `module` and `ldiskfs` packages.

   ```
   $ rpm -ivh kernel-lustre-smp-<ver> \
   kernel-ib-<ver> \
   lustre-modules-<ver> \
   lustre-ldiskfs-<ver>
   ```

b. **Install the utilities/userspace packages.**

   Use the `rpm -ivh` command to install the utilities packages. For example:

   ```
   $ rpm -ivh lustre-<ver>
   ```

c. **Install the e2fsprogs package.**

Make sure the e2fsprogs package downloaded in Step 4 is unpacked, and use the `rpm -i` command to install it. For example:

```
$ rpm -i e2fsprogs-<ver>
```

d. **(Optional) If you want to add optional packages to your Lustre system, install them now.**

5. **Verify that the boot loader (grub.conf or lilo.conf) has been updated to load the patched kernel.**

6. **Reboot the patched clients and the servers.**

a. **If you applied the patched kernel to any clients, reboot them.**

Unpatched clients do not need to be rebooted.

b. **Reboot the servers.**

Once all the machines have rebooted, the next steps are to configure Lustre Networking (LNET) and the Lustre file system. See Configuring Lustre.

3.3.3 Installing Lustre with a Third-Party Network Stack

When using third-party network hardware, you must follow a specific process to install and recompile Lustre. This section provides an installation example, describing how to install Lustre 1.6.6 while using the Myricom MX 1.2.7 driver. The same process is used for other third-party network stacks, by replacing MX-specific references in Step 2 with the stack-specific build and using the proper `--with` option when configuring the Lustre source code.

1. **Compile and install the Lustre kernel.**

a. **Install the necessary build tools. GCC and related tools must also be installed. For more information, see Required Lustre Software.**

```
$ yum install rpm-build redhat-rpm-config
$ mkdir -p rpmbuild/{BUILD,RPMS,SOURCES,SPECS,SRPMS}
$ echo '%_topdir %(echo $HOME)/rpmbuild' > .rpmmacros
```

b. **Install the patched Lustre source code.**

This RPM is available at the Lustre download page.

```
$ rpm -ivh kernel-lustre-source-2.6.18-92.1.10.el5_lustre.1.6.6.x86_64.rpm
```

c. Build the Linux kernel RPM.

```
$ cd /usr/src/linux-2.6.18-92.1.10.el5_lustre.1.6.6
$ make distclean
$ make oldconfig dep bzImage modules
$ cp /boot/config-`uname -r` .config
$ make oldconfig || make menuconfig
$ make include/asm
$ make include/linux/version.h
$ make SUBDIRS=scripts
$ make rpm
```

d. Install the Linux kernel RPM.

If you are building a set of RPMs for a cluster installation, this step is not necessary. Source RPMs are only needed on the build machine.

```
$ rpm -ivh ~/rpmbuild/kernel-lustre-2.6.18-92.1.10.el5_lustre.1.6.6.x86_64.rpm
$ mkinitrd /boot/2.6.18-92.1.10.el5_lustre.1.6.6
```

e. Update the boot loader (/etc/grub.conf) with the new kernel boot information.

```
$ /sbin/shutdown 0 -r
```

2. Compile and install the MX stack.

```
$ cd /usr/src/
$ gunzip mx_1.2.7.tar.gz (can be obtained from www.myri.com/scs/)
$ tar -xvf mx_1.2.7.tar
$ cd mx-1.2.7
$ ln -s common include
$ ./configure --with-kernel-lib
$ make
$ make install
```

3. **Compile and install the Lustre source code.**

   a. **Install the Lustre source (this can be done via RPM or tarball). The source file is available at the Lustre download page. This example shows installation via the tarball.**

   ```
   $ cd /usr/src/
   $ gunzip lustre-1.6.6.tar.gz
   $ tar -xvf lustre-1.6.6.tar
   ```

   b. **Configure and build the Lustre source code.**

   The `./configure --help` command shows a list of all of the `--with` options. All third-party network stacks are built in this manner.

   ```
   $ cd lustre-1.6.6
   $ ./configure --with-linux=/usr/src/linux --with-mx=/usr/src/mx-1.2.7
   $ make
   $ make rpms
   ```

   The `make rpms` command output shows the location of the generated RPMs

4. **Use the `rpm -ivh` command to install the RPMS.**

   ```
   $ rpm -ivh lustre-1.6.6-2.6.18_92.1.10.el5_lustre.1.6.6smp.x86_64.rpm
   $ rpm -ivh lustre-modules-1.6.6-2.6.18_92.1.10.el5_lustre.1.6.6smp.x86_64.rpm
   $ rpm -ivh lustre-ldiskfs-3.0.6-2.6.18_92.1.10.el5_lustre.1.6.6smp.x86_64.rpm
   ```

5. **Add the following lines to the /etc/modprobe.conf file.**

   ```
   options kmxlnd hosts=/etc/hosts.mxlnd
   options lnet networks=mx0(myri0),tcp0(eth0)
   ```

6. **Populate the myri0 configuration with the proper IP addresses.**

   ```
   vim /etc/sysconfig/network-scripts/myri0
   ```

7. **Add the following line to the /etc/hosts.mxlnd file.**

   ```
   $ IP HOST BOARD EP_ID
   ```

8. **Start Lustre.**

Once all the machines have rebooted, the next steps are to configure Lustre Networking (LNET) and the Lustre file system. See Configuring Lustre.

CHAPTER 4

Configuring Lustre

You can use the administrative utilities provided with Lustre to set up a system with many different configurations. This chapter shows how to configure a simple Lustre system comprised of a combined MGS/MDT, an OST and a client, and includes the following sections:

4.1 Configuring the Lustre File System

A Lustre file system consists of four types of subsystems – a Management Server (MGS), a Metadata Target (MDT), Object Storage Targets (OSTs) and clients. We recommend running these components on different systems, although, technically, they can co-exist on a single system. Together, the OSSs and MDS present a Logical Object Volume (LOV) which is an abstraction that appears in the configuration.

It is possible to set up the Lustre system with many different configurations by using the administrative utilities provided with Lustre. Some sample scripts are included in the directory where Lustre is installed. If you have installed the Lustre source code, the scripts are located in the `lustre/tests` sub-directory. These scripts enable quick setup of some simple, standard Lustre configurations.

Note – We recommend that you use dotted-quad IP addressing (IPv4) rather than host names. This aids in reading debug logs, and helps greatly when debugging configurations with multiple interfaces.

---

1. **Define the module options for Lustre networking (LNET), by adding this line to the `/etc/modprobe.conf` file[1].**

   ```
   options lnet networks=<network interfaces that LNET can use>
   ```

   This step restricts LNET to use only the specified network interfaces and prevents LNET from using all network interfaces.

   As an alternative to modifying the `modprobe.conf` file, you can modify the `modprobe.local` file or the configuration files in the `modprobe.d` directory.

Note – For details on configuring networking and LNET, see Configuring LNET.

---

2. **(Optional) Prepare the block devices to be used as OSTs or MDTs.**

   Depending on the hardware used in the MDS and OSS nodes, you may want to set up a hardware or software RAID to increase the reliability of the Lustre system. For more details on how to set up a hardware or software RAID, see the documentation for your RAID controller or see Lustre Software RAID Support.

1. The modprobe.conf file is a Linux file that lives in /etc/modprobe.conf and specifies what parts of the kernel are loaded.

3. **Create a combined MGS/MDT file system.**

   a. **Consider the MDT size needed to support the file system.**

      When calculating the MDT size, the only important factor is the number of files to be stored in the file system. This determines the number of inodes needed, which drives the MDT sizing. For more information, see Sizing the MDT and Planning for Inodes. Make sure the MDT is properly sized before performing the next step, as a too-small MDT can cause the space on the OSTs to be unusable.

   b. **Create the MGS/MDT file system on the block device. On the MDS node, run:**

      ```
      mkfs.lustre --fsname=<fsname> --mgs --mdt <block device name>
      ```

      The default file system name (fsname) is `lustre`.

Note – If you plan to generate multiple file systems, the MGS should be on its own dedicated block device.

---

4. **Mount the combined MGS/MDT file system on the block device. On the MDS node, run:**

   ```
   mount -t lustre <block device name> <mount point>
   ```

5. **Create the OST[2]. On the OSS node, run:**

   ```
   mkfs.lustre --ost --fsname=<fsname> --mgsnode=<NID> <block device name>
   ```

   You can have as many OSTs per OSS as the hardware or drivers allow.

   You should only use only 1 OST per block device. Optionally, you can create an OST which uses the raw block device and does not require partitioning.

Note – If the block device has more than 8 TB[3] of storage, it must be partitioned (because of the ext3 file system limitation). Lustre can support block devices with multiple partitions, but they are not recommended because of resulting bottlenecks.

---

6. **Mount the OST. On the OSS node where the OST was created, run:**

   ```
   mount -t lustre <block device name> <mount point>
   ```

2. When you create the OST, you are defining a storage device ('sd'), a device number (a, b, c, d), and a partition (1, 2, 3) where the OST node lives.

3. In Lustre 1.8.2, 16 TB on RHEL and 8 TB on other distributions.

Note – To create additional OSTs, repeat Step 4 and Step 5.

7. **Create the client (mount the file system on the client). On the client node, run:**

```
mount -t lustre <MGS node>:/<fsname> <mount point>
```

Note – To create additional clients, repeat Step 7.

8. **Verify that the file system started and is working correctly by running the `df`, `dd` and `ls` commands on the client node.**

   a. **Run the `lfs df -h` command.**

   ```
   [root@client1 /] lfs df -h
   ```

   The `lfs df -h` command lists space usage per OST and the MDT in human-readable format.

   b. **Run the `lfs df -ih` command.**

   ```
   [root@client1 /] lfs df -ih
   ```

   The `lfs df -ih` command lists inode usage per OST and the MDT.

   c. **Run the `dd` command.**

   ```
   [root@client1 /] cd /lustre
   ```

   ```
   [root@client1 /lustre] dd if=/dev/zero of=/lustre/zero.dat bs=4M
   count=2
   ```

   The `dd` command verifies write functionality by creating a file containing all zeros (0s). In this command, an 8 MB file is created.

   d. **Run the `ls` command.**

   ```
   [root@client1 /lustre] ls -lsah
   ```

   The `ls -lsah` command lists files and directories in the current working directory.

If you have a problem mounting the file system, check the syslogs for errors.

Tip – Now that you have configured Lustre, you can collect and register your service tags. For more information, see Service Tags.

4.1.0.1 Simple Lustre Configuration Example

To see the steps in a simple Lustre configuration, follow this worked example in which a combined MGS/MDT and two OSTs are created. Three block devices are used, one for the combined MGS/MDS node and one for each OSS node. Common parameters used in the example are listed below, along with individual node parameters.

Common Parameters	Value	Description
MGS node	`10.2.0.1@tcp0`	Node for the combined MGS/MDS
file system	`temp`	Name of the Lustre file system
network type	`TCP/IP`	Network type used for Lustre file system `temp`

Node Parameters	Value	Description
MGS/MDS node		
MGS/MDS node	`mdt1`	MDS in Lustre file system `temp`
block device	`/dev/sdb`	Block device for the combined MGS/MDS node
mount point	`/mnt/mdt`	Mount point for the `mdt1` block device (`/dev/sdb`) on the MGS/MDS node
First OSS node		
OSS node	`oss1`	First OSS node in Lustre file system `temp`
OST	`ost1`	First OST in Lustre file system `temp`
block device	`/dev/sdc`	Block device for the first OSS node (`oss1`)
mount point	`/mnt/ost1`	Mount point for the `ost1` block device (`/dev/sdc`) on the `oss1` node
Second OSS node		
OSS node	`oss2`	Second OSS node in Lustre file system `temp`
OST	`ost2`	Second OST in Lustre file system `temp`
block device	`/dev/sdd`	Block device for the second OSS node (`oss2`)
mount point	`/mnt/ost2`	Mount point for the `ost2` block device (`/dev/sdd`) on the `oss2` node
Client node		
client node	`client1`	Client in Lustre file system `temp`
mount point	`/lustre`	Mount point for Lustre file system `temp` on the `client1` node

1. **Define the module options for Lustre networking (LNET), by adding this line to the `/etc/modprobe.conf` file.**

```
options lnet networks=tcp
```

2. **Create a combined MGS/MDT file system on the block device. On the MDS node, run:**

```
[root@mds /]# mkfs.lustre --fsname=temp --mgs --mdt /dev/sdb
```

This command generates this output:

```
            Permanent disk data:
Target:             temp-MDTffff
Index:              unassigned
Lustre FS:          temp
Mount type:         ldiskfs
Flags:              0x75
            (MDT MGS needs_index first_time update )
Persistent mount opts: errors=remount-ro,iopen_nopriv,user_xattr
Parameters: mdt.group_upcall=/usr/sbin/l_getgroups

checking for existing Lustre data: not found
device size = 16MB
2 6 18
formatting backing filesystem ldiskfs on /dev/sdb
            target nametemp-MDTffff
            4k blocks  0
            options    -i 4096 -I 512 -q -O dir_index,uninit_groups -F
mkfs_cmd = mkfs.ext2 -j -b 4096 -L temp-MDTffff  -i 4096 -I 512 -q -O
dir_index,uninit_groups -F /dev/sdb
Writing CONFIGS/mountdata
```

3. **Mount the combined MGS/MDT file system on the block device. On the MDS node, run:**

```
[root@mds /]# mount -t lustre /dev/sdb /mnt/mdt
```

This command generates this output:

```
Lustre: temp-MDT0000: new disk, initializing
Lustre: 3009:0:(lproc_mds.c:262:lprocfs_wr_group_upcall()) \
temp-MDT0000: group upcall set to /usr/sbin/l_getgroups
Lustre: temp-MDT0000.mdt: set parameter \
group_upcall=/usr/sbin/l_getgroups
Lustre: Server temp-MDT0000 on device /dev/sdb has started
```

4. **Create the OSTs.**

   In this example, the OSTs (ost1 and ost2) are being created or different OSSs (oss1 and oss2).

   a. Create ost1. On oss1 node, run:

```
[root@oss1 /]# mkfs.lustre --ost --fsname=temp --mgsnode=
10.2.0.1@tcp0 /dev/sdc
```

   The command generates this output:

```
            Permanent disk data:
Target:             temp-OSTffff
Index:              unassigned
Lustre FS:          temp
Mount type:         ldiskfs
Flags:              0x72
(OST needs_index first_time update)
Persistent mount opts: errors=remount-ro,extents,mballoc
Parameters: mgsnode=10.2.0.1@tcp

checking for existing Lustre data: not found
device size = 16MB
2 6 18
formatting backing filesystem ldiskfs on /dev/sdc
            target name   temp-OSTffff
            4k blocks     0
            options       -I 256 -q -O dir_index,uninit_groups -F
mkfs_cmd = mkfs.ext2 -j -b 4096 -L temp-OSTffff  -I 256 -q -O
dir_index,uninit_groups -F /dev/sdc
Writing CONFIGS/mountdata
```

   b. Create ost2. On oss2 node, run:

```
[root@oss2 /]# mkfs.lustre --ost --fsname=temp --mgsnode=
10.2.0.1@tcp0 /dev/sdd
```

   The command generates this output:

```
            Permanent disk data:
Target:         temp-OSTffff
Index:          unassigned
Lustre FS:      temp
Mount type:     ldiskfs
Flags:          0x72
(OST needs_index first_time update)
Persistent mount opts: errors=remount-ro,extents,mballoc
Parameters: mgsnode=10.2.0.1@tcp
```

```
checking for existing Lustre data: not found
device size = 16MB
2 6 18
formatting backing filesystem ldiskfs on /dev/sdd
       target name    temp-OSTffff
       4k blocks      0
       options        -I 256 -q -O dir_index,uninit_groups -F
mkfs_cmd = mkfs.ext2 -j -b 4096 -L temp-OSTffff  -I 256 -q -O
dir_index,uninit_groups -F /dev/sdc
Writing CONFIGS/mountdata
```

5. **Mount the OSTs.**

   Mount each OST (ost1 and ost2), on the OSS where the OST was created.

   a. Mount ost1. On oss1 node, run:

   ```
   root@oss1 /] mount -t lustre /dev/sdc /mnt/ost1
   ```

   The command generates this output:

   ```
   LDISKFS-fs: file extents enabled
   LDISKFS-fs: mballoc enabled
   Lustre: temp-OST0000: new disk, initializing
   Lustre: Server temp-OST0000 on device /dev/sdb has started
   ```

   Shortly afterwards, this output appears:

   ```
   Lustre: temp-OST0000: received MDS connection from 10.2.0.1@tcp0
   Lustre: MDS temp-MDT0000: temp-OST0000_UUID now active, resetting
   orphans
   ```

   b. Mount ost2. On oss2 node, run:

   ```
   root@oss2 /] mount -t lustre /dev/sdd /mnt/ost2
   ```

   The command generates this output:

   ```
   LDISKFS-fs: file extents enabled
   LDISKFS-fs: mballoc enabled
   Lustre: temp-OST0000: new disk, initializing
   Lustre: Server temp-OST0000 on device /dev/sdb has started
   ```

   Shortly afterwards, this output appears:

   ```
   Lustre: temp-OST0000: received MDS connection from 10.2.0.1@tcp0
   Lustre: MDS temp-MDT0000: temp-OST0000_UUID now active, resetting
   orphans
   ```

6. **Create the client (mount the file system on the client). On the client node, run:**

```
root@client1 /] mount -t lustre 10.2.0.1@tcp0:/temp /lustre
```

This command generates this output:

```
Lustre: Client temp-client has started
```

7. **Verify that the file system started and is working by running the `df`, `dd` and `ls` commands on the client node.**

a. Run the `df` command:

```
[root@client1 /] lfs df -h
```

This command generates output similar to this:

```
Filesystem            Size   Used   Avail  Use%   Mounted on
/dev/mapper/VolGroup00-LogVol00
                      7.2G   2.4G   4.5G   35%    /
dev/sda1              99M    29M    65M    31%    /boot
tmpfs                 62M    0      62M    0%     /dev/shm
10.2.0.1@tcp0:/temp   30M    8.5M   20M    30%    /lustre
```

b. Run the `dd` command:

```
[root@client1 /] cd /lustre
[root@client1 /lustre] dd if=/dev/zero of=/lustre/zero.dat bs=4M
count=2
```

This command generates output similar to this:

```
2+0 records in
2+0 records out
8388608 bytes (8.4 MB) copied, 0.159628 seconds, 52.6 MB/s
```

c. Run the `ls` command:

```
[root@client1 /lustre] ls -lsah
```

This command generates output similar to this:

```
total 8.0M
4.0K drwxr-xr-x  2 root root 4.0K Oct 16 15:27 .
8.0K drwxr-xr-x 25 root root 4.0K Oct 16 15:27 ..
8.0M -rw-r--r--  1 root root 8.0M Oct 16 15:27 zero.dat
```

4.1.0.2 Module Setup

Make sure the modules (like LNET) are installed in the appropriate `/lib/modules` directory. The `mkfs.lustre` utility tries to automatically load LNET (via the Lustre module) with the default network settings (using all available network interfaces). To change this default setting, use the `network=...` option to specify the network(s) that LNET should use:

```
modprobe -v lustre "networks=XXX"
```

For example, to load Lustre with multiple-interface support (meaning LNET will use more than one physical circuit for communication between nodes), load the Lustre module with the following `network=...` option:

```
modprobe -v lustre "networks=tcp0(eth0),o2ib0(ib0)"
```

where:

`tcp0` is the network itself (TCP/IP)

`eth0` is the physical device (card) that is used (Ethernet)

`o2ib0` is the interconnect (InfiniBand)

4.1.1 Scaling the Lustre File System

A Lustre file system can be scaled by adding OSTs or clients. For instructions on creating additional OSTs see Step 4 and Step 5 above; for clients, see Step 7.

4.2 Additional Lustre Configuration

Once the Lustre file system is configured, it is ready for use. If additional configuration is necessary, several configuration utilities are available. For man pages and reference information, see:

- mkfs.lustre
- tunefs.lustre
- lctl
- mount.lustre

System Configuration Utilities (man8) profiles utilities (e.g., lustre_rmmod, e2scan, l_getgroups, llobdstat, llstat, plot-llstat, routerstat, and ll_recover_lost_found_objs), and tools to manage large clusters, perform application profiling, and debug Lustre.

4.3 Basic Lustre Administration

Once you have the Lustre file system up and running, you can use the procedures in this section to perform these basic Lustre administration tasks:

- Specifying the File System Name
- Starting up Lustre
- Mounting a Server
- Unmounting a Server
- Working with Inactive OSTs
- Finding Nodes in the Lustre File System
- Mounting a Server Without Lustre Service
- Specifying Failout/Failover Mode for OSTs
- Running Multiple Lustre File Systems
- Setting and Retrieving Lustre Parameters
- Regenerating the Lustre Configuration Logs
- Changing a Server NID
- Removing and Restoring OSTs
- Changing a Server NID
- Aborting Recovery
- Determining Which Machine is Serving an OST
- Failover
- Unmounting a Server (without Failover)
- Unmounting a Server (with Failover)
- Changing the Address of a Failover Node

4.3.1 Specifying the File System Name

The file system name is limited to 8 characters. We have encoded the file system and target information in the disk label, so you can mount by label. This allows system administrators to move disks around without worrying about issues such as SCSI disk reordering or getting the `/dev/device` wrong for a shared target. Soon, file system naming will be made as fail-safe as possible. Currently, Linux disk labels are limited to 16 characters. To identify the target within the file system, 8 characters are reserved, leaving 8 characters for the file system name:

`<fsname>-MDT0000` or `<fsname>-OST0a19`

To mount by label, use this command:

```
$ mount -t lustre -L <file system label> <mount point>
```

This is an example of mount-by-label:

```
$ mount -t lustre -L testfs-MDT0000 /mnt/mdt
```

Caution – Mount-by-label should NOT be used in a multi-path environment.

Although the file system name is internally limited to 8 characters, you can mount the clients at any mount point, so file system users are not subjected to short names. Here is an example:

```
mount -t lustre uml1@tcp0:/shortfs /mnt/<long-file_system-name>
```

4.3.2 Starting up Lustre

The startup order of Lustre components depends on whether you have a combined MGS/MDT or these components are separate.

- If you have a combined MGS/MDT, the recommended startup order is OSTs, then the MGS/MDT, and then clients.
- If the MGS and MDT are separate, the recommended startup order is: MGS, then OSTs, then the MDT, and then clients.

Note – If an OST is added to a Lustre file system with a combined MGS/MDT, then the startup order changes slightly; the MGS must be started first because the OST needs to write its configuration data to it. In this scenario, the startup order is MGS/MDT, then OSTs, then the clients.

4.3.3 Mounting a Server

Starting a Lustre server is straightforward and only involves the `mount` command. Lustre servers can be added to `/etc/fstab`:

```
mount -t lustre
```

The mount command generates output similar to this:

```
/dev/sda1 on /mnt/test/mdt type lustre (rw)
/dev/sda2 on /mnt/test/ost0 type lustre (rw)
192.168.0.21@tcp:/testfs on /mnt/testfs type lustre (rw)
```

In this example, the MDT, an OST (`ost0`) and file system (`testfs`) are mounted.

```
LABEL=testfs-MDT0000 /mnt/test/mdt lustre defaults,_netdev,noauto 0 0

LABEL=testfs-OST0000 /mnt/test/ost0 lustre defaults,_netdev,noauto 0 0
```

In general, it is wise to specify noauto and let your high-availability (HA) package manage when to mount the device. If you are not using failover, make sure that networking has been started before mounting a Lustre server. RedHat, SuSE, Debian (and perhaps others) use the `_netdev` flag to ensure that these disks are mounted after the network is up.

We are mounting by disk label here—the label of a device can be read with e2label. The label of a newly-formatted Lustre server ends in FFFF, meaning that it has yet to be assigned. The assignment takes place when the server is first started, and the disk label is updated.

Caution – Do not do this when the client and OSS are on the same node, as memory pressure between the client and OSS can lead to deadlocks.

Caution – Mount-by-label should NOT be used in a multi-path environment.

4.3.4 Unmounting a Server

To stop a Lustre server, use the `umount <mount point>` command.

For example, to stop `ost0` on mount point `/mnt/test`, run:

```
$ umount /mnt/test
```

Gracefully stopping a server with the `umount` command preserves the state of the connected clients. The next time the server is started, it waits for clients to reconnect, and then goes through the recovery procedure.

If the force (`-f`) flag is used, then the server evicts all clients and stops WITHOUT recovery. Upon restart, the server does not wait for recovery. Any currently connected clients receive I/O errors until they reconnect.

Note – If you are using loopback devices, use the `-d` flag. This flag cleans up loop devices and can always be safely specified.

4.3.5 Working with Inactive OSTs

To mount a client or an MDT with one or more inactive OSTs, run commands similar to this:

```
client> mount -o exclude=testfs-OST0000 -t lustre uml1:/testfs\
/mnt/testfs
client> cat /proc/fs/lustre/lov/testfs-clilov-*/target_obd
```

To activate an inactive OST on a live client or MDT, use the `lctl activate` command on the OSC device. For example:

```
lctl --device 7 activate
```

Note – A colon-separated list can also be specified. For example, `exclude=testfs-OST0000:testfs-OST0001`.

4.3.6 Finding Nodes in the Lustre File System

There may be situations in which you need to find all nodes in your Lustre file system or get the names of all OSTs.

To get a list of all Lustre nodes, run this command on the MGS:

```
# cat /proc/fs/lustre/mgs/MGS/live/*
```

Note – This command must be run on the MGS.

In this example, file system `lustre` has three nodes, `lustre-MDT0000`, `lustre-OST0000`, and `lustre-OST0001`.

```
cfs21:/tmp# cat /proc/fs/lustre/mgs/MGS/live/*
fsname: lustre
flags: 0x0     gen: 26
lustre-MDT0000
lustre-OST0000
lustre-OST0001
```

To get the names of all OSTs, run this command on the MDS:

```
# cat /proc/fs/lustre/lov/<fsname>-mdtlov/target_obd
```

Note – This command `must` be run on the MDS.

In this example, there are two OSTs, `lustre-OST0000` and `lustre-OST0001`, which are both active.

```
cfs21:/tmp# cat /proc/fs/lustre/lov/lustre-mdtlov/target_obd
0: lustre-OST0000_UUID ACTIVE
1: lustre-OST0001_UUID ACTIVE
```

4.3.7 Mounting a Server Without Lustre Service

If you are using a combined MGS/MDT, but you only want to start the MGS and not the MDT, run this command:

```
mount -t lustre <MDT partition> -o nosvc <mount point>
```

The `<MDT partition>` variable is the combined MGS/MDT.

In this example, the combined MGS/MDT is `testfs-MDT0000` and the mount point is `mnt/test/mdt`.

```
$ mount -t lustre -L testfs-MDT0000 -o nosvc /mnt/test/mdt
```

4.3.8 Specifying Failout/Failover Mode for OSTs

Lustre uses two modes, failout and failover, to handle an OST that has become unreachable because it fails, is taken off the network, is unmounted, etc.

- In **failout** mode, Lustre clients immediately receive errors (EIOs) after a timeout, instead of waiting for the OST to recover.
- In **failover** mode, Lustre clients wait for the OST to recover.

By default, the Lustre file system uses failover mode for OSTs. To specify failout mode instead, run this command:

```
$ mkfs.lustre --fsname=<fsname> --ost --mgsnode=<MGS node NID>
--param="failover.mode=failout" <block device name>
```

In this example, failout mode is specified for the OSTs on MGS `uml1`, file system `testfs`.

```
$ mkfs.lustre --fsname=testfs --ost --mgsnode=uml1 --param=
"failover.mode=failout" /dev/sdb
```

Caution – Before running this command, unmount all OSTS that will be affected by the change in the failover/failout mode.

Note – After initial file system configuration, use the `tunefs.lustre` utility to change the failover/failout mode. For example, to set the failout mode, run:

```
$ tunefs.lustre --param failover.mode=failout <OST partition>
```

4.3.9 Running Multiple Lustre File Systems

There may be situations in which you want to run multiple file systems. This is doable, as long as you follow specific naming conventions.

By default, the `mkfs.lustre` command creates a file system named `lustre`. To specify a different file system name (limited to 8 characters), run this command:

```
mkfs.lustre --fsname=<new file system name>
```

Note – The MDT, OSTs and clients in the new file system must share the same name (prepended to the device name). For example, for a new file system named `foo`, the MDT and two OSTs would be named `foo-MDT0000`, `foo-OST0000`, and `foo-OST0001`.

To mount a client on the file system, run:

```
mount -t lustre mgsnode:/<new fsname> <mountpoint>
```

For example, to mount a client on file system `foo` at mount point `/mnt/lustre1`, run:

```
mount -t lustre mgsnode:/foo /mnt/lustre1
```

Note – If a client(s) will be mounted on several file systems, add the following line to `/etc/xattr.conf` file to avoid problems when files are moved between the file systems: `lustre.* skip`

Note – The MGS is universal; there is only one MGS per Lustre installation, not per file system.

Note – There is only one file system per MDT. Therefore, specify `--mdt --mgs` on one file system and `--mdt --mgsnode=<MGS node NID>` on the other file systems.

A Lustre installation with two file systems (`foo` and `bar`) could look like this, where the MGS node is `mgsnode@tcp0` and the mount points are `/mnt/lustre1` and `/mnt/lustre2`.

```
mgsnode# mkfs.lustre --mgs /mnt/lustre1

mdtfoonode# mkfs.lustre --fsname=foo --mdt \
--mgsnode=mgsnode@tcp0 /mnt/lustre1

ossfoonode# mkfs.lustre --fsname=foo --ost \
--mgsnode=mgsnode@tcp0 /mnt/lustre1

ossfoonode# mkfs.lustre --fsname=foo --ost \
--mgsnode=mgsnode@tcp0 /mnt/lustre2

mdtbarnode# mkfs.lustre --fsname=bar --mdt \
--mgsnode=mgsnode@tcp0 /mnt/lustre1

ossbarnode# mkfs.lustre --fsname=bar --ost \
--mgsnode=mgsnode@tcp0 /mnt/lustre1

ossbarnode# mkfs.lustre --fsname=bar --ost \
--mgsnode=mgsnode@tcp0 /mnt/lustre2
```

To mount a client on file system `foo` at mount point `/mnt/lustre1`, run:

```
mount -t lustre mgsnode@tcp0:/foo /mnt/lustre1
```

To mount a client on file system `bar` at mount point `/mnt/lustre2`, run:

```
mount -t lustre mgsnode@tcp0:/bar /mnt/lustre2
```

4.3.10 Setting and Retrieving Lustre Parameters

There are several options for setting parameters in Lustre.

- When the file system is created, using `mkfs.lustre`. See Setting Parameters with mkfs.lustre
- When a server is stopped, using `tunefs.lustre`. See Setting Parameters with tunefs.lustre
- When the file system is running, using `lctl`. See Setting Parameters with lctl

Additionally, you can use `lctl` to retrieve Lustre parameters. See Reporting Current Parameter Values.

4.3.10.1 Setting Parameters with mkfs.lustre

When the file system is created, parameters can simply be added as a `--param` option to the `mkfs.lustre` command. For example:

```
$ mkfs.lustre --mdt --param="sys.timeout=50" /dev/sda
```

4.3.10.2 Setting Parameters with tunefs.lustre

If a server (OSS or MDS) is stopped, parameters can be added using the `--param` option to the `tunefs.lustre` command. For example:

```
$ tunefs.lustre --param="failover.node=192.168.0.13@tcp0" /dev/sda
```

With `tunefs.lustre`, parameters are "additive" -- new parameters are specified in addition to old parameters, they do not replace them. To erase all old `tunefs.lustre` parameters and just use newly-specified parameters, run:

```
$ tunefs.lustre --erase-params --param=<new parameters>
```

The `tunefs.lustre` command can be used to set any parameter settable in a /proc/fs/lustre file and that has its own OBD device, so it can be specified as `<obd|fsname>.<obdtype>.<proc_file_name>=<value>`. For example:

```
$ tunefs.lustre --param mdt.group_upcall=NONE /dev/sda1
```

4.3.10.3 Setting Parameters with lctl

When the file system is running, the `lctl` command can be used to set parameters (temporary or permanent) and report current parameter values. Temporary parameters are active as long as the server or client is not shut down. Permanent parameters live through server and client reboots.

Setting Temporary Parameters

Use the `lctl set_param` command to set temporary parameters on the node where it is run. These parameters map to items in `/proc/{fs,sys}/{lnet,lustre}`. The `lctl set_param` command uses this syntax:

```
lctl set_param [-n] <obdtype>.<obdname>.<proc_file_name>=<value>
```

For example:

```
# lctl set_param osc.*.max_dirty_mb=1024

osc.myth-OST0000-osc.max_dirty_mb=32
osc.myth-OST0001-osc.max_dirty_mb=32
osc.myth-OST0002-osc.max_dirty_mb=32
osc.myth-OST0003-osc.max_dirty_mb=32
osc.myth-OST0004-osc.max_dirty_mb=32
```

Setting Permanent Parameters

Use the `lctl conf_param` command to set permanent parameters. In general, the `lctl conf_param` command can be used to specify any parameter settable in a /proc/fs/lustre file, with its own OBD device. The `lctl conf_param` command uses this syntax (same as the `mkfs.lustre` and `tunefs.lustre` commands):

```
<obd|fsname>.<obdtype>.<proc_file_name>=<value>)
```

Here are a few examples of `lctl conf_param` commands:

```
$ mgs> lctl conf_param testfs-MDT0000.sys.timeout=40
$ lctl conf_param testfs-MDT0000.mdt.group_upcall=NONE
$ lctl conf_param testfs.llite.max_read_ahead_mb=16
$ lctl conf_param testfs-MDT0000.lov.stripesize=2M
$ lctl conf_param testfs-OST0000.osc.max_dirty_mb=29.15
$ lctl conf_param testfs-OST0000.ost.client_cache_seconds=15
$ lctl conf_param testfs.sys.timeout=40
```

Caution – Parameters specified with the `lctl conf_param` command are set permanently in the file system's configuration file on the MGS.

4.3.10.4 Reporting Current Parameter Values

To report current Lustre parameter values, use the `lctl get_param` command with this syntax:

```
lctl get_param [-n] <obdtype>.<obdname>.<proc_file_name>
```

This example reports data on RPC service times.

```
$ lctl get_param -n ost.*.ost_io.timeouts

  service : cur 1 worst 30 (at 1257150393, 85d23h58m54s ago) 1 1 1 1
```

This example reports the number of inodes available on each OST.

```
# lctl get_param osc.*.filesfree

osc.myth-OST0000-osc-ffff88006dd20000.filesfree=217623
osc.myth-OST0001-osc-ffff88006dd20000.filesfree=5075042
osc.myth-OST0002-osc-ffff88006dd20000.filesfree=3762034
osc.myth-OST0003-osc-ffff88006dd20000.filesfree=91052
osc.myth-OST0004-osc-ffff88006dd20000.filesfree=129651
```

4.3.11 Regenerating the Lustre Configuration Logs

If the Lustre system's configuration logs are in a state where the file system cannot be started, use the `writeconf` command to erase them. After the `writeconf` command is run and the servers restart, the configuration logs are re-generated and stored on the MGS (as in a new file system).

You should only use the `writeconf` command if:

- The configuration logs are in a state where the file system cannot start
- A server NID is being changed

The `writeconf` command is destructive to some configuration items (i.e., OST pools information and items set via `conf_param`), and should be used with caution. To avoid problems:

- Shut down the file system before running the `writeconf` command
- Run the `writeconf` command on all servers (MDT first, then OSTs)
- Start the file system in this order (MDT first, then OSTs, then clients)

Caution – Lustre 1.8 introduces the OST pools feature, which enables a group of OSTs to be named for file striping purposes. If you use OST pools, be aware that running the `writeconf` command erases **all** pools information (as well as any other parameters set via `lctl conf_param`). We recommend that the pools definitions (and `conf_param` settings) be executed via a script, so they can be reproduced easily after a `writeconf` is performed.

To regenerate the Lustre system's configuration logs:

1. **Shut down the file system by unmounting all servers and clients.**

   Do not run `writeconf` until the file system is completely shut down.

2. **Run the writeconf command on all servers.**

   Run writeconf on the MDT first, and then the OSTs.

   a. **On the MDT, run:**

   ```
   <mdt node>$ tunefs.lustre --writeconf <device>
   ```

   b. **On each OST, run:**

   ```
   <ost node>$ tunefs.lustre --writeconf <device>
   ```

3. **Restart the file system by mounting the MGS/MDT first (if co-located), then the OSTs, then the clients. If the MGS and MDT are separate, mount the MGS first, then the MDT, OSTs and clients.**

   After the `writeconf` command is run, the configuration logs are re-generated as servers restart.

4.3.12 Changing a Server NID

If you need to change the NID on the MDT or an OST, run the `writeconf` command to erase Lustre configuration information (including server NIDs), and then re-generate the system configuration using updated server NIDs.

Change a server NID in these situations:

- New server hardware is added to the file system, and the MDS or an OSS is being moved to the new machine
- New network card is installed in the server
- You want to reassign IP addresses

To change a server NID:

1. **Update the LNET configuration in the /etc/modprobe.conf file so the list of server NIDs (`lctl list_nids`) is correct.**

   The `lctl list_nids` command indicates which network(s) are configured to work with Lustre.

2. **Shut down the file system by unmounting all servers and clients.**

   Do not run the `writeconf` command until the file system is completely shut down.

3. **Run the writeconf command on all servers.**

   Run `writeconf` on the MDT first, and then the OSTs.

   a. **On the MDT, run:**

   ```
   $ mdt> tunefs.lustre --writeconf <device>
   ```

   b. **On each OST, run:**

   ```
   $ ost> tunefs.lustre --writeconf <device>
   ```

   c. **If the NID on the MGS was changed, communicate the new MGS location to each server. Run:**

   ```
   tunefs.lustre --erase-param --mgsnode=<new_nid(s)> --writeconf /dev/..
   ```

4. **Restart the file system by mounting the MGS/MDT first (if co-located), then the OSTs, then the clients. If the MGS and MDT are separate, mount the MGS first, then the MDT, OSTs and clients.**

   After the `writeconf` command is run, the configuration logs are re-generated as servers restart, and server NIDs in the updated `list_nids` file are used.

4.3.13 Removing and Restoring OSTs

OSTs can be removed from and restored to a Lustre file system. Currently in Lustre, removing an OST really means that the OST is 'deactivated' in the file system, not permanently removed. A removed OST still appears in the file system; do not create a new OST with the same name.

You may want to remove (deactivate) an OST and prevent new files from being written to it in several situations:

- Hard drive has failed and a RAID resync/rebuild is underway
- OST is nearing its space capacity

4.3.13.1 Removing an OST from the File System

When removing an OST, remember that the MDT does not communicate directly with OSTs. Rather, each OST has a corresponding OSC which communicates with the MDT. It is necessary to determine the device number of the OSC that corresponds to the OST. Then, you use this device number to deactivate the OSC on the MDT.

To remove an OST from the file system:

1. **For the OST to be removed, determine the device number of the corresponding OSC on the MDT.**

   a. **List all OSCs on the node, along with their device numbers. Run:**

   ```
   lctl dl | grep " osc "
   ```

   This is sample `lctl dl | grep " osc "` output:

   ```
   11 UP osc lustre-OST-0000-osc-cac94211 4ea5b30f-6a8e-55a0-7519-2f20318ebdb4 5
   12 UP osc lustre-OST-0001-osc-cac94211 4ea5b30f-6a8e-55a0-7519-2f20318ebdb4 5
   13 IN osc lustre-OST-0000-osc lustre-MDT0000-mdtlov_UUID 5
   14 UP osc lustre-OST-0001-osc lustre-MDT0000-mdtlov_UUID 5
   ```

   b. **Determine the device number of the OSC that corresponds to the OST to be removed.**

2. **Temporarily deactivate the OSC on the MDT. On the MDT, run:**

   ```
   $ mdt> lctl --device <devno> deactivate
   ```

   For example, based on the command output in Step 1, to deactivate device 13 (the MDT's OSC for OST-0000), the command would be:

   ```
   $ mdt> lctl --device 13 deactivate
   ```

   This marks the OST as inactive on the MDS, so no new objects are assigned to the OST. This does not prevent use of existing objects for reads or writes.

Note – Do not deactivate the OST on the clients. Do so causes errors (EIOs), and the copy out to fail.

Caution – Do not use `lctl conf_param` to deactivate the OST. It permanently sets a parameter in the file system configuration.

---

3. **Discover all files that have objects residing on the deactivated OST. Run:**

   ```
   lfs find --obd {OST UUID} / <mount_point>
   ```

4. **Copy (not move) the files to a new directory in the file system.**

   Copying the files forces object re-creation on the active OSTs.

5. **Move (not copy) the files back to their original directory in the file system.**

   Moving the files causes the original files to be deleted, as the copies replace them.

6. **Once all files have been moved, permanently deactivate the OST on the clients and the MDT. On the MGS, run:**

   ```
   # mgs> lctl conf_param <OST name>.osc.active=0
   ```

Temporarily Deactivating an OST in the File System

You may encounter situations when it is necessary to temporarily deactivate an OST, rather than permanently deactivate it. For example, you may need to deactivate a failed OST that cannot be immediately repaired, but want to continue to access the remaining files on the available OSTs.

To temporarily deactivate an OST:

1. **Mount the Lustre file system.**
2. **On the MDS and all clients, run:**

```
# lctl set_param osc.<faname>-<OST name>-*.active=0
```

Clients accessing files on the deactivated OST receive an IO error (-5), rather than pausing until the OST completes recovery.

4.3.13.2 Restoring an OST in the File System

Restoring an OST to the file system is as easy as activating it. When the OST is active, it is automatically added to the normal stripe rotation and files are written to it.

To restore an OST:

1. **Make sure the OST to be restored is running.**
2. **Reactivate the OST. On the MGS, run:**

```
# mgs> lctl conf_param <OST name>.osc.active=1
```

4.3.14 Aborting Recovery

You can abort recovery with either the lctl utility or by mounting the target with the abort_recov option (mount -o abort_recov). When starting a target, run:

```
$ mount -t lustre -L <MDT name> -o abort_recov <mount point>
```

Note – The recovery process is blocked until all OSTs are available.

4.3.15 Determining Which Machine is Serving an OST

In the course of administering a Lustre file system, you may need to determine which machine is serving a specific OST. It is not as simple as identifying the machine's IP address, as IP is only one of several networking protocols that Lustre uses and, as such, LNET does not use IP addresses as node identifiers, but NIDs instead.

To identify the NID that is serving a specific OST, run one of the following commands on a client (you do not need to be a root user):

```
client$ lctl get_param osc.${fsname}-${OSTname}*.ost_conn_uuid
```

For example:

```
client$ lctl get_param osc.*-OST0000*.ost_conn_uuid
osc.myth-OST0000-osc-f1579000.ost_conn_uuid=192.168.20.1@tcp
```

- OR -

```
client$ lctl get_param osc.*.ost_conn_uuid
osc.myth-OST0000-osc-f1579000.ost_conn_uuid=192.168.20.1@tcp
osc.myth-OST0001-osc-f1579000.ost_conn_uuid=192.168.20.1@tcp
osc.myth-OST0002-osc-f1579000.ost_conn_uuid=192.168.20.1@tcp
osc.myth-OST0003-osc-f1579000.ost_conn_uuid=192.168.20.1@tcp
osc.myth-OST0004-osc-f1579000.ost_conn_uuid=192.168.20.1@tcp
```

4.4 More Complex Configurations

If a node has multiple network interfaces, it may have multiple NIDs. When a node is specified, all of its NIDs must be listed, delimited by commas (,) so other nodes can choose the NID that is appropriate for their network interfaces. When failover nodes are specified, they are delimited by a colon (:) or by repeating a keyword (`--mgsnode=` or `--failnode=`). To obtain all NIDs from a node (while LNET is running), run:

```
lctl list_nids
```

This displays the server's NIDs (networks configured to work with Lustre).

4.4.1 Failover

This example has a combined MGS/MDT failover pair on uml1 and uml2, and a OST failover pair on uml3 and uml4. There are corresponding Elan addresses on uml1 and uml2.

```
uml1> mkfs.lustre --fsname=testfs --mdt --mgs \
--failnode=uml2,2@elan /dev/sda1
uml1> mount -t lustre /dev/sda1 /mnt/test/mdt
uml3> mkfs.lustre --fsname=testfs --ost --failnode=uml4 \
--mgsnode=uml1,1@elan --mgsnode=uml2,2@elan /dev/sdb
uml3> mount -t lustre /dev/sdb /mnt/test/ost0
client> mount -t lustre uml1,1@elan:uml2,2@elan:/testfs /mnt/testfs
uml1> umount /mnt/mdt
uml2> mount -t lustre /dev/sda1 /mnt/test/mdt
uml2> cat /proc/fs/lustre/mds/testfs-MDT0000/recovery_status
```

Where multiple NIDs are specified, comma-separation (for example, `uml2,2@elan`) means that the two NIDs refer to the same host, and that Lustre needs to choose the "best" one for communication. Colon-separation (for example, `uml1:uml2`) means that the two NIDs refer to two different hosts, and should be treated as failover locations (Lustre tries the first one, and if that fails, it tries the second one.)

Note – If you have an MGS or MDT configured for failover, perform these steps:

1. On the OST, list the NIDs of all MGS nodes at mkfs time.

```
OST# mkfs.lustre --fsname sunfs --ost --mgsnode=10.0.0.1
--mgsnode=10.0.0.2 /dev/{device}
```

2. On the client, mount the file system.

```
client# mount -t lustre 10.0.0.1:10.0.0.2:/sunfs /cfs/client/
```

4.5 Operational Scenarios

In the operational scenarios below, the management node is the MDS. The management service is started as the initial part of the startup of the primary MDT.

Tip – All targets that are configured for failover must have some kind of shared storage among two server nodes.

IP Network, Single MDS, Single OST, No Failover

On the MDS, run:

```
mkfs.lustre --mdt --mgs --fsname=<fsname> <partition>
mount -t lustre <partition> <mountpoint>
```

On the OSS, run:

```
mkfs.lustre --ost --mgs --fsname=<fsname> <partition>
mount -t lustre <partition> <mountpoint>
```

On the client, run:

```
mount -t lustre <MGS NID>:/<fsname> <mountpoint>
```

IP Network, Failover MDS

For failover, storage holding target data must be available as shared storage to failover server nodes. Failover nodes are statically configured as mount options.

On the MDS, run:

```
mkfs.lustre --mdt --mgs --fsname=<fsname> \
--failover=<failover MGS NID> <partition>
mount -t lustre <partition> <mount point>
```

On the OSS, run:

```
mkfs.lustre --ost --mgs --fsname=<fsname> \
--mgsnode=<MGS NID>,<failover MGS NID> <partition>
mount -t lustre <partition> <mount point>
```

On the client, run:

```
mount -t lustre <MGS NID>[,<failover MGS NID>]:/<fsname> \
<mount point>
```

IP Network, Failover MDS and OSS

On the MDS, run:

```
mkfs.lustre --mdt --mgs --fsname=<fsname> \
--failover=<failover MGS NID> <partition>
mount -t lustre <partition> <mount point>
```

On the OSS, run:

```
mkfs.lustre --ost --mgs --fsname=<fsname> \
--mgsnode=<MGS NID>[,<failover mds hostdesc>] \
--failover=<failover OSS NID> <partition>
mount -t lustre <partition> <mount point>
```

On the client, run:

```
mount -t lustre <MGS NID>[,<failover MGS NID>]:/<fsname> \
<mount point>
```

4.5.1 Unmounting a Server (without Failover)

To stop a server (MDS or OSS) without failover, run:

```
umount <mds|oss mountpoint>
```

This stops the server unconditionally, and cleans up client connections and export information. When the server restarts, the clients create a new connection to it.

4.5.2 Unmounting a Server (with Failover)

To stop a server (MDS or OSS) with failover, run:

```
umount -f <MDS|OSS mount point>
```

This stops the server and preserves client export information. When the server restarts, the clients reconnect and resume in-progress transactions.

4.5.3 Changing the Address of a Failover Node

To change the address of a failover node (e.g, to use node X instead of node Y), run this command on the OSS/OST partition:

```
tunefs.lustre --erase-params --failnode=<NID> <device>
```

CHAPTER 5

Service Tags

This chapter describes the use of service tags with Lustre, and includes the following sections:

5.1 Introduction to Service Tags

Service tags are part of an IT asset inventory management system provided by Sun. A service tag is a unique identifier for a piece of hardware or software (gear) that enables usage data about the tagged item to be shared over a local network in standard XML format. The service tag program is used for a number of Sun products, including hardware, software and services, and has now been implemented for Lustre.

Service tags are provided for each MGS, MDS, OSS node and Lustre client. Using service tags enables automatic discovery and tracking of these system components, so administrators can better manage their Lustre environment.

Note – Service tags are used solely to provide an inventory list of system and software information to Sun; they do not contain any personal information. Service tag components that communicate information are read-only and contained. They are not capable of accepting information and they cannot communicate with any other services on your system.

For more information on service tags, see the Service Tag wiki and Service Tag FAQ.

5.2 Using Service Tags

To begin using service tags with your Lustre system, download the service tag package and registration client. The entire service tag process can be easily managed from the Sun Inventory webpage.

5.2.1 Installing Service Tags

Service tag packages (for RedHat and SuSE Linux) are downloadable from the Sun Lustre downloads page. To download and install the service tags package:

1. **Navigate to the Sun Lustre download page and download the service tag package, sun-servicetag-1.1.4-1.i386.rpm[1], for Lustre.**
2. **Install the service tag package on all Lustre nodes (MGSs, MDSs, OSSs and clients).**

   The service tag package includes several init.d scripts which are started on reboot (`/etc/init.d/stosreg` and `/etc/init.d/psn start`).

   This package also adds entries in the [x]inetd's configuration scripts to provide remote access to the nodes needed to collect information. The script restarts [x]inetd (`killall -HUP xinetd 1>/dev/null 2>&1`).
3. **If this is a new installation, format the OSTs, MDTs, MGSs and Lustre clients.**
4. **Mount the OSTs, MDTs, MGSs and Lustre clients, and verify that the Lustre file system is running normally.**

1. This is the current service tag package. The version number is subject to change.

5.2.2 Discovering and Registering Lustre Components

After installing the service tag package on all of your Lustre nodes, discover and register the Lustre components. To perform this procedure, Lustre must be fully configured and running.

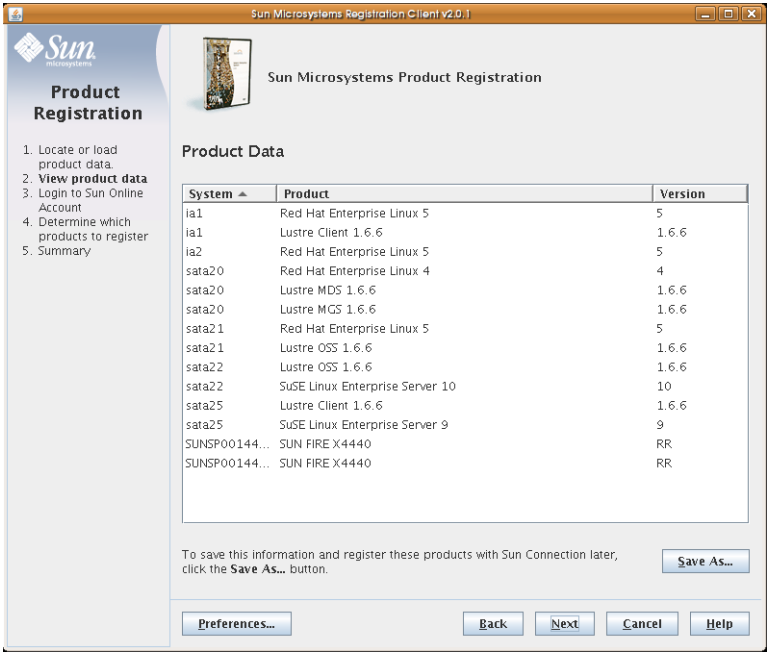
1. **Navigate to the Sun Lustre download page and download the Registration client, eis-regclient.jar.**
2. **Install the Registration client on one node (the collection node) that can reach all Lustre clients and servers over a TCP/IP network.**
3. **Install Java Virtual Machine (Java VM) on the collection node.**

   Java VM is available at the Sun Java download site.
4. **Start the Registration client, run:**

   ```
   $ java -jar eis-regclient.jar
   ```

   The Registration Client utility launches.

FIGURE 5-1 Registration Client



Note – The Registration client requires an X display to run. If the node from which you want to do the registration has no native X display, you can use SSH's X forwarding to display the Registration client interface on your local machine.

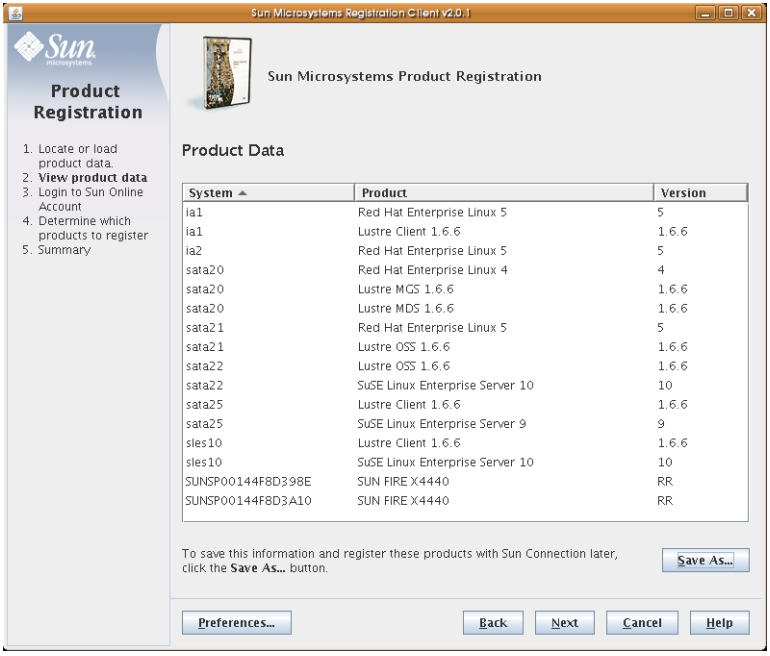
The registration process includes up to five steps. The first step is to discover the service tags created when you started Lustre.

The Registration client looks for Sun products on your local subnet, by default. Alternately, you can specify another subnet, specific hosts or IP addresses.

5. **Select an option to locate service tags and click Next.**

   The Product Data screen displays Sun products (that support service tags) as they are located. For each product, the system name, product name, and version (if applicable) are listed.

FIGURE 5-2 Product Data



If the list of located products does not look complete, select **Back** and enter a more accurate search.

Note – Located service tags are not limited to Lustre components. The Registration client locates any Sun product on your system that is supported in the Sun inventory management program.

6. **Register the service tags or save them for later use.**

   There are two options for registering service tags.

   - Click **Next** to continue with the remaining steps 3-5 of the registration process, including authentication to the Inventory management website and uploading your service tags.
   - Save the collected service tags and register them on another machine. This option is good if the system used to collect the service tags does not have Web access. Click **Save As** and enter a file where the tags should be saved. You can then move this file (using network copy, a USB key, etc.) to a machine with Web access.

     On the Web-access machine, navigate to Sun Inventory and click **Discover & Register** to start the Registration client. Select the 'Locate Product on Other Subnets, Specific System or Load Previously Saved Data' option and check the 'File Name' box. Enter (or navigate to) the file where the collected service tags were saved, click **Next** and follow the remaining steps 3-5 to complete the registration process, including authentication to the Inventory management website and uploading your service tags.

7. **If you wish, navigate to** Sun Inventory **and log into your account to view and manage your IT assets.**

Note – For more information about service tags, see https://inventory.sun.com, which links to the http://wikis.sun.com/display/ServiceTag/Home wiki. This wiki includes an FAQ about Sun's service tag program.

5.2.3 Information Registered with Sun

The service tag registration process collects the following product, registration agentry and system information.

Data Name	Description
Product Information	
Lustre-specific information	Node type (client, MDS, OSS or MGS)
Instance identifier	Unique identifier for that instance of the gear
Product name	Name of the gear
Product identifier	Unique identifier for the gear being registered
Product vendor	Vendor of the gear
Product version	Version of the gear
Parent name	Parent gear of the registered gear
Parent identifier	Unique identifier for the parent of the gear
Customer tag	Optional, customer-defined value
Time stamp	Day and time that the gear is registered
Source	Where the gear identifiers came from
Container	Name of the gear's container
Registration Agentry Information	
Agentry Identifier	Unique value for that instance of the agentry
Agentry Version	Value of the agentry
Registry Identifier	File version containing product registration information
System Information	
Host	System hostname
System	Operating System
Release	Operating system version
Architecture	Physical hardware architecture
Platform	Hardware platform
Manufacturer	Hardware manufacturer
CPU manufacturer	CPU manufacturer
HostID	System host ID
Serial number	System chassis serial number

CHAPTER 6

Configuring Lustre - Examples

This chapter provides Lustre configuration examples and includes the following section:

6.1 Simple TCP Network

This chapter presents several examples of Lustre configurations on a simple TCP network.

6.1.1 Lustre with Combined MGS/MDT

Below is an example is of a Lustre setup “datafs” having combined MDT/MGS with four OSTs and a number of Lustre clients.

6.1.1.1 Installation Summary

- Combined (co-located) MDT/MGS
- Four OSTs
- Any number of Lustre clients

6.1.1.2 Configuration Generation and Application

1. **Install the Lustre RPMS (per Installing Lustre) on all nodes that are going to be part of the Lustre file system. Boot the nodes in Lustre kernel, including the clients.**

2. **Change modprobe.conf by adding the following line to it.**

```
options lnet networks=tcp
```

3. **Configuring Lustre on MGS and MDT node.**

```
$ mkfs.lustre --fsname datafs --mdt --mgs /dev/sda
```

4. **Make a mount point on MDT/MGS for the file system and mount it.**

```
$ mkdir -p /mnt/data/mdt
$ mount -t lustre /dev/sda /mnt/data/mdt
```

5. **Configuring Lustre on all four OSTs.**

```
mkfs.lustre --fsname datafs --ost --mgsnode=mds16@tcp0 /dev/sda
mkfs.lustre --fsname datafs --ost --mgsnode=mds16@tcp0 /dev/sdd
mkfs.lustre --fsname datafs --ost --mgsnode=mds16@tcp0 /dev/sda1
mkfs.lustre --fsname datafs --ost --mgsnode=mds16@tcp0 /dev/sdb
```

Note – While creating the file system, make sure you are not using disk with the operating system.

---

6. **Make a mount point on all the OSTs for the file system and mount it.**

```
$ mkdir -p /mnt/data/ost0
$ mount -t lustre /dev/sda /mnt/data/ost0

$ mkdir -p /mnt/data/ost1
$ mount -t lustre /dev/sdd /mnt/data/ost1

$ mkdir -p /mnt/data/ost2
$ mount -t lustre /dev/sda1 /mnt/data/ost2

$ mkdir -p /mnt/data/ost3
$ mount -t lustre /dev/sdb /mnt/data/ost3

$ mount -t lustre mdt16@tcp0:/datafs /mnt/datafs
```

6.1.2 Lustre with Separate MGS and MDT

The following example describes a Lustre file system “datafs” having an MGS and an MDT on separate nodes, four OSTs, and a number of Lustre clients.

6.1.2.1 Installation Summary

- One MGS
- One MDT
- Four OSTs
- Any number of Lustre clients

6.1.2.2 Configuration Generation and Application

1. **Install the Lustre RPMs (per Installing Lustre) on all the nodes that are going to be a part of the Lustre file system. Boot the nodes in the Lustre kernel, including the clients.**
2. **Change the modprobe.conf by adding the following line to it.**

```
options lnet networks=tcp
```

3. **Start Lustre on the MGS node.**

```
$ mkfs.lustre --mgs /dev/sda
```

4. **Make a mount point on MGS for the file system and mount it.**

```
$ mkdir -p /mnt/mgs
$ mount -t lustre /dev/sda1 /mnt/mgs
```

5. **Start Lustre on the MDT node.**

```
$ mkfs.lustre --fsname=datafs --mdt --mgsnode=mgsnode@tcp0 \
/dev/sda2
```

6. **Make a mount point on MDT/MGS for the file system and mount it.**

```
$ mkdir -p /mnt/data/mdt
$ mount -t lustre /dev/sda /mnt/data/mdt
```

7. **Start Lustre on all the four OSTs.**

```
mkfs.lustre --fsname datafs --ost --mgsnode=mds16@tcp0 /dev/sda
mkfs.lustre --fsname datafs --ost --mgsnode=mds16@tcp0 /dev/sdd
mkfs.lustre --fsname datafs --ost --mgsnode=mds16@tcp0 /dev/sda1
mkfs.lustre --fsname datafs --ost --mgsnode=mds16@tcp0 /dev/sdb
```

8. **Make a mount point on all the OSTs for the file system and mount it**

```
$ mkdir -p /mnt/data/ost0
$ mount -t lustre /dev/sda /mnt/data/ost0

$ mkdir -p /mnt/data/ost1
$ mount -t lustre /dev/sdd /mnt/data/ost1

$ mkdir -p /mnt/data/ost2
$ mount -t lustre /dev/sda1 /mnt/data/ost2

$ mkdir -p /mnt/data/ost3
$ mount -t lustre /dev/sdb /mnt/data/ost3

$ mount -t lustre mdsnode@tcp0:/datafs /mnt/datafs
```

6.1.2.3 Configuring Lustre with a CSV File

A new utility (script) - `/usr/sbin/lustre_config` can be used to configure Lustre 1.6 and later. This script enables you to automate formatting and setup of disks on multiple nodes.

Describe your entire installation in a Comma Separated Values (CSV) file and pass it to the script. The script contacts multiple Lustre targets simultaneously, formats the drives, updates `modprobe.conf`, and produces HA configuration files using definitions in the CSV file. (The `lustre_config -h` option shows several samples of CSV files.)

Note – The CSV file format is a file type that stores tabular data. Many popular spreadsheet programs, such as Microsoft Excel, can read from/write to CSV files.

How lustre_config Works

The `lustre_config` script parses each line in the CSV file and executes remote commands, like `mkfs.lustre`, to format each Lustre target in the Lustre cluster.

Optionally, the `lustre_config` script can also:

- Verify network connectivity and hostnames in the cluster
- Configure Linux MD/LVM devices
- Modify /etc/modprobe.conf to add Lustre networking information
- Add the Lustre server information to `/etc/fstab`
- Produce configurations for Heartbeat or CluManager

How to Create a CSV File

Five different types of line formats are available to create a CSV file. Each line format represents a target. The list of targets with the respective line formats are described below:

Linux MD device

The CSV line format is:

hostname, MD, md name, operation mode, options, raid level, component devices

Where:

Variable	Supported Type
hostname	Hostname of the node in the cluster.
MD	Marker of the MD device line.
md name	MD device name, for example: `/dev/md0`
operation mode	Operations mode, either **create** or **remove**. Default is **create**.
options	A 'catchall' for other mdadm options, for example, `-c 128`
raid level	RAID level: 0, 1, 4, 5, 6, 10, linear and multipath.
hostname	Hostname of the node in the cluster.
component devices	Block devices to be combined into the MD device. Multiple devices are separated by space or by using shell extensions, for example: `/dev/sd{a,b,c}`

Linux LVM PV (Physical Volume)

The CSV line format is:

hostname, PV, pv names, operation mode, options

Where:

Variable	Supported Type
hostname	Hostname of the node in the cluster.
PV	Marker of the PV line.
pv names	Devices or loopback files to be initialized for later use by LVM or to wipe the label, for example: `/dev/sda` Multiple devices or files are separated by space or by using shell expansions, for example: `/dev/sd{a,b,c}`
operation mode	Operations mode, either **create** or **remove**. Default is **create**.
options	A 'catchall' for other pvcreate/pvremove options, for example: `-vv`

Linux LVM VG (Volume Group)

The CSV line format is:

hostname, VG, vg name, operation mode, options, pv paths

Where:

Variable	Supported Type
hostname	Hostname of the node in the cluster.
VG	Marker of the VG line.
vg name	Name of the volume group, for example: `ost_vg`
operation mode	Operations mode, either **create** or **remove**. Default is **create**.
options	A 'catchall' for other vgcreate/rgremove options, for example: `-s 32M`
pv paths	Physical volumes to construct this VG, required by the create mode; multiple PVs are separated by space or by using shell expansions, for example: `/dev/sd[k-m]1`

Linux LVM LV (Logical Volume)

The CSV line format is:

hostname, LV, lv name, operation mode, options, lv size, vg name

Where:

Variable	Supported Type
hostname	Hostname of the node in the cluster.
LV	Marker of the LV line.
lv name	Name of the logical volume to be created (optional) or path of the logical volume to be removed (required by the remove mode).
operation mode	Operations mode, either **create** or **remove**. Default is **create**.
options	A 'catchall' for other lvcreate/lvremove options, for example: `-i 2 -l 128`
lv size	Size [kKmMgGtT] to be allocated for the new LV. Default is megabytes (MB).
vg name	Name of the VG in which the new LV is created.

Lustre target

The CSV line format is:

hostname, module_opts, device name, mount point, device type, fsname, mgs nids, index, format options, mkfs options, mount options, failover nids

Where:

Variable	Supported Type
hostname	Hostname of the node in the cluster. It must match `uname -n`
module_opts	Lustre networking module options. Use the newline character (\n) to delimit multiple options.
device name	Lustre target (block device or loopback file).
mount point	Lustre target mount point.
device type	Lustre target type (mgs, mdt, ost, mgs|mdt, mdt|mgs).
fsname	Lustre file system name (limit is 8 characters).
mgs nids	NID(s) of the remote mgs node, required for MDT and OST targets; if this item is not given for an MDT, it is assumed that the MDT is also an MGS (according to mkfs.lustre).
index	Lustre target index.
format options	A 'catchall' contains options to be passed to mkfs.lustre. For example: `device-size`, `--param`, and so on.
mkfs options	Format options to be wrapped with `--mkfsoptions=` and passed to mkfs.lustre.
mount options	If this script is invoked with `-m` option, then the value of this item is wrapped with `--mountfsoptions=` and passed to mkfs.lustre; otherwise, the value is added into `/etc/ fstab`
failver nids	NID(s) of the failover partner node.

Note – In one node, all NIDs are delimited by commas (','). To use comma-separated NIDs in a CSV file, they must be enclosed in quotation marks, for example: `"lustre-mgs2,2@elan"`

When multiple nodes are specified, they are delimited by a colon (':').

If you leave a blank, it is set to default.

The `lustre_config.csv` file looks like:

```
{mdtname}.{domainname},options lnet networks=
tcp,/dev/sdb,/mnt/mdt,mgs|mdt
{ost2name}.{domainname},options lnet networks=
tcp,/dev/sda,/mnt/ost1,ost,,192.168.16.34@tcp0
{ost1name}.{domainname},options lnet networks=
tcp,/dev/sda,/mnt/ost0,ost,,192.168.16.34@tcp0
```

Note – Provide a Fully Qualified Domain Name (FQDN) for all nodes that are a part of the file system in the first parameter of all the rows starting in a new line. For example:

```
mdt1.clusterfs.com,options lnet networks=
tcp,/dev/sdb,/mnt/mdt,mgs|mdt
```

- AND -

```
ost1.clusterfs.com,options lnet\ networks=tcp,/dev/sda,/mnt/
ost1,ost,,192.168.16.34@tcp0
```

Using CSV with lustre_config

Once you created the CSV file, you can start to configure the file system by using the `lustre_config` script.

1. List the available parameters. At the command prompt. Type:

```
$ lustre_config
lustre_config: Missing csv file!

Usage: lustre_config [options] <csv file>

  This script is used to format and set up multiple lustre servers
  from a csv file.

  Options:
  -h       help and examples
  -a       select all the nodes from the csv file to operate on
  -w       hostname,hostname,...

  select the specified list of nodes (separated by commas) to
  operate on rather than all the nodes in the csv file

  -x       hostname,hostname,... exclude the specified list of
  nodes (separated by commas)
  -t       HAtype produce High-Availability software
  configurations

  The argument following -t is used to indicate the
  High-Availability software type. The HA software types which are
  currently supported are: hbv1 (Heartbeat version 1) and hbv2
  (Heartbeat version 2).

  -n       no net - don't verify network connectivity and hostnames
  in the cluster
  -d       configure Linux MD/LVM devices before formatting the
  Lustre targets
  -f       force-format the Lustre targets using --reformat option
  OR you can specify --reformat in the ninth field of the target
  line in the csv file
  -m       no fstab change - don't modify /etc/fstab to add the new
  Lustre targets. If using this option, then the value of "mount
  options" item in the csv file will be passed to mkfs.lustre,else
  the value will be added into the /etc/fstab
  -v       verbose mode
  csv file is a spreadsheet that contains configuration parameters
  (separated by commas) for each target in a Lustre cluster
```

Example 1: Simple Lustre configuration with CSV (use the following command):

```
$ lustre_config -v -a -f lustre_config.csv
```

This command starts the execution and configuration on the nodes or targets in `lustre_config.csv`, prompting you for the password to log in with root access to the nodes. To avoid this prompt, configure a shell like pdsh or SSH.

After completing the above steps, the script makes Lustre target entries in the `/etc/fstab` file on Lustre server nodes, such as:

```
/dev/sdb          /mnt/mdtlustre  defaults      0   0

/dev/sda          /mnt/ostlustre  defaults      0   0
```

2. **Run** `mount /dev/sdb` **and** `mount /dev/sda` **to start the Lustre services.**

Note – Use the `/usr/sbin/lustre_createcsv` script to collect information on Lustre targets from running a Lustre cluster and generating a CSV file. It is a reverse utility (compared to `lustre_config`) and should be run on the MGS node.

Example 2: More complicated Lustre configuration with CSV:

For RAID and LVM-based configuration, the `lustre_config.csv` file looks like this:

```
# Configuring RAID 5 on mds16.clusterfs.com
mds16.clusterfs.com,MD,/dev/md0,,-c 128,5,/dev/sdb /dev/sdc
/dev/sdd

# configuring multiple RAID5 on oss161.clusterfs.com
oss161.clusterfs.com,MD,/dev/md0,,-c 128,5,/dev/sdb /dev/sdc
/dev/sdd
oss161.clusterfs.com,MD,/dev/md1,,-c 128,5,/dev/sde /dev/sdf
/dev/sdg

# configuring LVM2-PV from the RAID5 from the above steps on
oss161.clusterfs.com
oss161.clusterfs.com,PV,/dev/md0 /dev/md1

# configuring LVM2-VG from the PV and RAID5 from the above steps on
oss161.clusterfs.com
oss161.clusterfs.com,VG,oss_data,,-s 32M,/dev/md0 /dev/md1

# configuring LVM2-LV from the VG, PV and RAID5 from the above steps
on oss161.clusterfs.com
oss161.clusterfs.com,LV,ost0,,-i 2 -I 128,2G,oss_data
oss161.clusterfs.com,LV,ost1,,-i 2 -I 128,2G,oss_data
```

```
# configuring LVM2-PV on oss162.clusterfs.com
oss162.clusterfs.com,PV, /dev/sdb /dev/sdc /dev/sdd /dev/sde
/dev/sdf /dev/sdg

# configuring LVM2-VG from the PV from the above steps on
oss162.clusterfs.com
oss162.clusterfs.com,VG,vg_oss1,,-s 32M,/dev/sdb /dev/sdc /dev/sdd
oss162.clusterfs.com,VG,vg_oss2,,-s 32M,/dev/sde /dev/sdf /dev/sdg

# configuring LVM2-LV from the VG and PV from the above steps on
oss162.clusterfs.com
oss162.clusterfs.com,LV,ost3,,-i 3 -I 64,1G,vg_oss2
oss162.clusterfs.com,LV,ost2,,-i 3 -I 64,1G,vg_oss1

#configuring Lustre file system on MDS/MGS, OSS and OST with RAID
and LVM created above
mds16.clusterfs.com,options lnet networks=
tcp,/dev/md0,/mnt/mdt,mgs|mdt,,,,,,,
oss161.clusterfs.com,options lnet networks=
tcp,/dev/oss_data/ost0,/mnt/ost0,ost,,192.168.16.34@tcp0,,,,
oss161.clusterfs.com,options lnet networks=
tcp,/dev/oss_data/ost1,/mnt/ost1,ost,,192.168.16.34@tcp0,,,,
oss162.clusterfs.com,options lnet networks=
tcp,/dev/pv_oss1/ost2,/mnt/ost2,ost,,192.168.16.34@tcp0,,,,
oss162.clusterfs.com,options lnet networks=
tcp,/dev/pv_oss2/ost3,/mnt/ost3,ost,,192.168.16.34@tcp0,,,,
$ lustre_config -v -a -d -f lustre_config.csv
```

This command creates RAID and LVM, and then configures Lustre on the nodes or targets specified in `lustre_config.csv`. The script prompts you for the password to log in with root access to the nodes.

After completing the above steps, the script makes Lustre target entries in the `/etc/fstab` file on Lustre server nodes, such as:

For MDS | MDT:

```
/dev/md0 /mnt/mdtlustre defaults00
```

For OSS:

```
/pv_oss1/ost2 /mnt/ost2lustre defaults00
```

3. **Start the Lustre services, run:**

```
mount /dev/sdb
mount /dev/sda
```

CHAPTER 7

More Complicated Configurations

This chapter describes more complicated Lustre configurations and includes the following sections:

7.1 Multihomed Servers

If you are using multiple networks with Lustre, certain configuration settings are required. Throughout this section, a worked example is used to illustrate these settings.

In this example, servers `megan` and `oscar` each have three TCP NICs (eth0, eth1, and eth2) and an Elan NIC. The eth2 NIC is used for management purposes and should not be used by LNET. TCP clients have a single TCP interface and Elan clients have a single Elan interface.

7.1.1 Modprobe.conf

Options under `modprobe.conf` are used to specify the networks available to a node. You have the choice of two different options – the `networks` option, which explicitly lists the networks available and the `ip2nets` option, which provides a list-matching lookup. Only one option can be used at any one time. The order of LNET lines in `modprobe.conf` is important when configuring multi-homed servers. If a server node can be reached using more than one network, the first network specified in `modprobe.conf` will be used.

Networks

On the servers:

```
options lnet networks=tcp0(eth0, eth1),elan0
```

Elan-only clients:

```
options lnet networks=elan0
```

```
TCP-only clients:
```

```
options lnet networks=tcp0
```

Note – In the case of TCP-only clients, the first available non-loopback IP interface is used for tcp0 since the interfaces are not specified.

---

ip2nets

The `ip2nets` option is typically used to provide a single, universal modprobe.conf file that can be run on all servers and clients. An individual node identifies the locally available networks based on the listed IP address patterns that match the node's local IP addresses. Note that the IP address patterns listed in the `ip2nets` option are only used to identify the networks that an individual node should instantiate. They are not used by LNET for any other communications purpose. The servers `megan` and `oscar` have eth0 IP addresses 192.168.0.2 and .4. They also have IP over Elan (eip) addresses of 132.6.1.2 and .4. TCP clients have IP addresses 192.168.0.5-255. Elan clients have eip addresses of 132.6.[2-3].2, .4, .6, .8.

`modprobe.conf` is identical on all nodes:

```
options lnet 'ip2nets="tcp0(eth0,eth1)192.168.0.[2,4]; tcp0 \
192.168.0.*; elan0 132.6.[1-3].[2-8/2]"'
```

Note – LNET lines in `modprobe.conf` are only used by the local node to determine what to call its interfaces. They are not used for routing decisions.

Because `megan` and `oscar` match the first rule, LNET uses eth0 and eth1 for tcp0 on those machines. Although they also match the second rule, it is the first matching rule for a particular network that is used. The servers also match the (only) Elan rule. The [2-8/2] format matches the range 2-8 stepping by 2; that is 2,4,6,8. For example, clients at 132.6.3.5 would not find a matching Elan network.

7.1.2 Start Servers

For the combined MGS/MDT with TCP network, run:

```
$ mkfs.lustre --fsname spfs --mdt --mgs /dev/sda
$ mkdir -p /mnt/test/mdt
$ mount -t lustre /dev/sda /mnt/test/mdt
```

- OR -

For the MGS on the separate node with TCP network, run:

```
$ mkfs.lustre --mgs /dev/sda
$ mkdir -p /mnt/mgs
$ mount -t lustre /dev/sda /mnt/mgs
```

For starting the MDT on node mds16 with MGS on node mgs16, run:

```
$ mkfs.lustre --fsname=spfs --mdt --mgsnode=mgs16@tcp0 /dev/sda
$ mkdir -p /mnt/test/mdt
$ mount -t lustre /dev/sda2 /mnt/test/mdt
```

For starting the OST on TCP-based network, run:

```
$ mkfs.lustre --fsname spfs --ost --mgsnode=mgs16@tcp0 /dev/sda$
$ mkdir -p /mnt/test/ost0
$ mount -t lustre /dev/sda /mnt/test/ost0
```

7.1.3 Start Clients

TCP clients can use the host name or IP address of the MDS, run:

```
mount -t lustre megan@tcp0:/mdsA/client /mnt/lustre
```

Use this command to start the Elan clients, run:

```
mount -t lustre 2@elan0:/mdsA/client /mnt/lustre
```

Note – If the MGS node has multiple interfaces (for instance, cfs21 and 1@elan), only the client mount command has to change. The MGS NID specifier must be an appropriate nettype for the client (for example, a TCP client could use uml1@tcp0, and an Elan client could use 1@elan). Alternatively, a list of all MGS NIDs can be given, and the client chooses the correctd one. For example:

```
$ mount -t lustre mgs16@tcp0,1@elan:/testfs /mnt/testfs
```

7.2 Elan to TCP Routing

Servers `megan` and `oscar` are on the Elan network with eip addresses 132.6.1.2 and .4. Megan is also on the TCP network at 192.168.0.2 and routes between TCP and Elan. There is also a standalone router, router1, at Elan 132.6.1.10 and TCP 192.168.0.10. Clients are on either Elan or TCP.

7.2.1 Modprobe.conf

`modprobe.conf` is identical on all nodes, run:

```
options lnet 'ip2nets="tcp0 192.168.0.*; elan0 132.6.1.*"' \
'routes="tcp [2,10]@elan0; elan 192.168.0.[2,10]@tcp0"'
```

7.2.2 Start servers

To start router1, run:

```
modprobe lnet
lctl network configure
```

To start megan and oscar, run:

```
$ mkfs.lustre --fsname spfs --mdt --mgs /dev/sda
$ mkdir -p /mnt/test/mdt
$ mount -t lustre /dev/sda /mnt/test/mdt
$ mount -t lustre mgs16@tcp0,1@elan:/testfs /mnt/testfs
```

7.2.3 Start clients

For the TCP client, run:

```
mount -t lustre megan:/mdsA/client /mnt/lustre/
```

For the Elan client, run:

```
mount -t lustre 2@elan0:/mdsA/client /mnt/lustre
```

7.3 Load Balancing with InfiniBand

A Lustre file system contains OSSs with two InfiniBand HCAs. Lustre clients have only one InfiniBand HCA using OFED Infiniband "o2ib" drivers. Load balancing between the HCAs on the OSS is accomplished through LNET.

7.3.1 Setting Up modprobe.conf for Load Balancing

To configure LNET for load balancing on clients and servers:

1. **Set the modprobe.conf options.**

   Depending on your configuration, set modprobe.conf options as follows:

   - Dual HCA OSS server

     ```
     options lnet ip2nets= "o2ib0(ib0),o2ib1(ib1) 192.168.10.1.[101-102]
     ```

   - Client with the odd IP address

     ```
     options lnet ip2nets=o2ib0(ib0) 192.168.10.[103-253/2]
     ```

   - Client with the even IP address

     ```
     options lnet ip2nets=o2ib1(ib0) 192.168.10.[102-254/2]
     ```

2. **Start the servers (MGS, MDT and OSS).**

   ```
   modprobe lnet
   ```

   ```
   $ mkfs.lustre --fsname lustre --mdt --mgs /dev/sda
   $ mkdir -p /mnt/test/mdt
   $ mount -t lustre /dev/sda /mnt/test/mdt
   $ mount -t lustre mgs@o2ib0:/lustre /mnt/mdt
   ```

   ```
   $ mkfs.lustre --fsname lustre --ost --mgsnode=mds@o2ib0 /dev/sda
   $ mkdir -p /mnt/test/mdt
   $ mount -t lustre /dev/sda /mnt/test/ost
   $ mount -t lustre mgs@o2ib0:/lustre /mnt/ost
   ```

3. **Start the clients.**

   This example shows the IB client being mounted.

   ```
   mount -t lustre
   192.168.10.101@o2ib0,192.168.10.102@o2ib1:/mds/client /mnt/lustre
   ```

7.4 Multi-Rail Configurations with LNET

To aggregate bandwidth across both rails of a dual-rail IB cluster (o2iblnd)[1] using LNET, consider these points:

- LNET can work with multiple rails, however, it does not load balance across them. The actual rail used for any communication is determined by the peer NID.
- Multi-rail LNET configurations do not provide an additional level of network fault tolerance. The configurations described below are for bandwidth aggregation only. Network interface failover is planned as an upcoming Lustre feature.
- A Lustre node always uses the same local NID to communicate with a given peer NID. The criteria used to determine the local NID are:
  - Fewest hops (to minimize routing), and
  - Appears first in the "networks" or "ip2nets" LNET configuration strings

As an example, consider a two-rail IB cluster running the OFA stack (OFED) with these IPoIB address assignments.

```
             ib0                 ib1
Servers      192.168.0.*         192.168.1.*
Clients      192.168.[2-127].*   192.168.[128-253].*
```

1. Multi-rail configurations are only supported by o2iblnd; other IB LNDs do not support multiple interfaces.

You could create these configurations:

- A cluster with more clients than servers. The fact that an individual client cannot get two rails of bandwidth is unimportant because the servers are the actual bottleneck.

```
ip2nets="o2ib0(ib0),  o2ib1(ib1)192.168.[0-1].*           #all servers;\
                      o2ib0(ib0) 192.168.[2-253].[0-252/2]#even clients;\
                      o2ib1(ib1) 192.168.[2-253].[1-253/2]#odd clients"
```

This configuration gives every server two NIDs, one on each network, and statically load-balances clients between the rails.

- A single client that must get two rails of bandwidth, and it does not matter if the maximum aggregate bandwidth is only (# servers) * (1 rail).

```
ip2nets="  o2ib0(ib0)             192.168.[0-1].[0-252/2]  #even servers;\
           o2ib1(ib1)             192.168.[0-1].[1-253/2]  #odd servers;\
           o2ib0(ib0),o2ib1(ib1) 192.168.[2-253].*         #clients"
```

This configuration gives every server a single NID on one rail or the other. Clients have a NID on both rails.

- All clients and all servers must get two rails of bandwidth.

```
ip2nets="  o2ib0(ib0),o2ib2(ib1) 192.168.[0-1].[0-252/2]  #even servers;\
           o2ib1(ib0),o2ib3(ib1) 192.168.[0-1].[1-253/2]  #odd servers;\
           o2ib0(ib0),o2ib3(ib1) 192.168.[2-253].[0-252/2)#even clients;\
           o2ib1(ib0),o2ib2(ib1) 192.168.[2-253].[1-253/2)#odd clients"
```

This configuration includes two additional proxy o2ib networks to work around Lustre's simplistic NID selection algorithm. It connects "even" clients to "even" servers with o2ib0 on rail0, and "odd" servers with o2ib3 on rail1. Similarly, it connects "odd" clients to "odd" servers with o2ib1 on rail0, and "even" servers with o2ib2 on rail1.

CHAPTER 8

Failover

This chapter describes failover in a Lustre system and includes the following sections:

8.1 What is Failover?

A computer system is "highly available" when the services it provides are available with minimal downtime. In a highly-available system, if a failure condition occurs, such as the loss of a server or a network or software fault, the system's services continue without interruption. Generally, we measure availability by the percentage of time the system is required to be available.

Availability is accomplished by replicating hardware and/or software so that when a primary server fails or is unavailable, a standby server can be switched into its place to run applications and associated resources. This process, called *failover*, should be automatic and, in most cases, completely application-transparent.

A failover hardware setup requires a pair of servers with a shared resource (typically a physical storage device, which may be based on SAN, NAS, hardware RAID, SCSI or FC technology). The method of sharing storage should be essentially transparent at the device level in that the same physical logical unit number (LUN) should be visible from both servers. To ensure high availability at the physical storage level, we encourage the use of RAID arrays to protect against drive-level failures.

8.1.1 Failover Capabilities

To establish a highly-available Lustre file system, power management software or hardware and high availability (HA) software are used to provide the following failover capabilities:

- **Resource fencing** - Protects physical storage from simultaneous access by two nodes.
- **Resource management** - Starts and stops the Lustre resources as a part of failover, maintains the cluster state, and carries out other resource management tasks.
- **Health monitoring** - Verifies the availability of hardware and network resources and responds to health indications provided by Lustre.

Although these capabilities can be provided by a variety of software and/or hardware solutions, the currently supported solution for Lustre is Heartbeat. For information about accessing the latest version of Heartbeat, see:

www.sun.com/software/products/hpcsoftware/getit.jsp

HA software is responsible for detecting failure of the primary Lustre server node and controlling the failover. Lustre works with any HA software that supports resource (I/O) fencing. For proper resource fencing, the HA software must be able to completely power off the failed server or disconnect it from the shared storage device. If two active nodes have access to the same storage device, data may be severely corrupted.

8.1.2 Types of Failover Configurations

Nodes in a cluster can be configured for failover in several ways. They are often configured in pairs (for example, two OSTs attached to a shared storage device), but other failover configurations are also possible. Failover configurations include:

- **Active/passive pair** - In this configuration, the active node provides resources and serves data, while the passive node is usually standing by idle. If the active node fails, the passive node takes over and becomes active.
- **Active/active pair** - In this configuration, both nodes are active, each providing a subset of resources. In case of a failure, the second node takes over resources from the failed node.

The active/passive configuration is seldom used for OST servers as it doubles hardware costs without improving performance. On the other hand, an active/active cluster configuration can improve performance by serving and providing arbitrary failover protection to a number of OSTs.

8.2 Failover Functionality in Lustre

The failover functionality provided in Lustre supports the following failover scenario. When a client attempts to do I/O to a failed Lustre target, it continues to try until it receives an answer from any of the configured failover nodes for the Lustre target. A user-space application does not detect anything unusual, except that the I/O may take longer than usual to complete.

Lustre failover requires two nodes configured as a failover pair, which must share one or more storage devices. Lustre can be configured to provide MDT or OST failover.

- For MDT failover, two MDSs are configured to serve the same MDT. Only one MDS node can serve an MDT at a time.
- For OST failover, multiple OSS nodes are configured to be able to serve the same OST. However, only one OSS node can serve the OST at a time. An OST can be moved between OSS nodes that have access to the same storage device using `umount/mount` commands.

To add a failover partner to a Lustre configuration, the `--failnode` option is used. This can be done at creation time (using `mkfs.lustre`) or later when the Lustre system is active (using `tunefs.lustre`). For explanations of these utilities, see mkfs.lustre and tunefs.lustre.

For a failover example, see More Complicated Configurations.

Note – Failover is supported in Lustre only at the file system level. In a complete failover solution, support for system-level components, such as node failure detection or power control, is provided by a third party tool.

Caution – OST failover functionality does not protect against corruption caused by a disk failure. If the storage media (i.e., physical disk) used for an OST fails, Lustre cannot recover it. We strongly recommend that some form of RAID be used for OSTs. Lustre functionality assumes that the storage is reliable, so it adds no extra reliability features.

8.2.1 MDT Failover Configuration (Active/Passive)

Two MDSs are usually configured as an active/passive failover pair. Note that both nodes must have access to shared storage for the MDT(s) and the MGS. The primary (active) MDS manages the Lustre system metadata resources. If the primary MDS fails, the secondary (passive) MDS takes over these resources and serves the MDTs and the MGS.

Note – In an environment with multiple file systems, the MDSs can be configured in a quasi active/active configuration, with each MDS managing metadata for a subset of the Lustre file system.

8.2.2 OST Failover Configuration (Active/Active)

OSTs are usually configured in a load-balanced, active/active failover configuration. A failover cluster is built from two OSSs.

Note – OSSs configured as a failover pair must have shared disks/RAID.

In an active configuration, 50% of the available OSTs are assigned to one OSS and the remaining OSTs are assigned to the other OSS. Each OSS serves as the primary node for half the OSTs and as a failover node for the remaining OSTs.

In this mode, if one OSS fails, the other OSS takes over all of the failed OSTs. The clients attempt to connect to each OSS serving the OST, until one of them responds. Data on the OST is written synchronously, and the clients replay transactions that were in progress and uncommitted to disk before the OST failure.

8.2.3 Lustre Failover and MMP

The failover functionality in Lustre is supported by the multiple mount protection (MMP) feature, which protects the file system from being mounted simultaneously to more than one node. This feature is important in a shared storage environment (for example, when a failover pair of OSTs share a partition).

Lustre's backend file system, `ldiskfs`, supports the MMP mechanism. A block in the file system is updated by a `kmmpd` daemon at one second intervals, and a sequence number is written in this block. If the file system is cleanly unmounted, then a special "clean" sequence is written to this block. When mounting the file system, `ldiskfs` checks if the MMP block has a clean sequence or not.

Even if the MMP block has a clean sequence, `ldiskfs` waits for some interval to guard against the following situations:

- If I/O traffic is heavy, it may take longer for the MMP block to be updated.
- If another node is trying to mount the same file system, a "race" condition may occur.

With MMP enabled, mounting a clean file system takes at least 10 seconds. If the file system was not cleanly unmounted, then the file system mount may require additional time.

Note – The MMP feature is only supported on Linux kernel versions >= 2.6.9.

8.2.3.1 Working with MMP

On a new Lustre file system, MMP is automatically enabled by `mkfs.lustre` at format time if failover is being used and the kernel and e2fsprogs version support it. On an existing file system, a Lustre administrator can manually enable MMP when the file system is unmounted.

Use the following commands to determine whether MMP is running in Lustre and to enable or disable the MMP feature.

To determine if MMP is enabled, run:

```
dumpe2fs -h <device>|grep mmp
```

Here is a sample command:

```
dumpe2fs -h /dev/sdc | grep mmp
Filesystem features: has_journal ext_attr resize_inode dir_index
filetype extent mmp sparse_super large_file uninit_bg
```

To manually disable MMP, run:

```
tune2fs -O ^mmp <device>
```

To manually enable MMP, run:

```
tune2fs -O mmp <device>
```

When MMP is enabled, if `ldiskfs` detects multiple mount attempts after the file system is mounted, it blocks these later mount attempts and reports the time when the MMP block was last updated, the node name, and the device name of the node where the file system is currently mounted.

8.3 Configuring and Using Heartbeat with Lustre Failover

This section describes how to configure Lustre failover using the Heartbeat cluster infrastructure daemon.

8.3.1 Creating a Failover Environment

Lustre provides failover mechanisms only at the file system level. No failover support is provided for system-level components, such as node failure detection or power control, as would typically be provided in a complete failover solution. Additional tools are also needed to provide resource fencing, control and monitoring.

8.3.1.1 Power Management Software

Lustre failover requires power control and management capability to verify that a failed node is shut down before I/O is directed to the failover node. This avoids double-mounting the two nodes, and the risk of unrecoverable data corruption. A variety of power management tools will work, but two packages that are commonly used with Lustre are STONITH and PowerMan.

Shoot The Other Node In The HEAD (STONITH), is a set of power management tools provided with the Linux-HA package. STONITH has native support for many power control devices and is extensible. It uses expect scripts to automate control.

PowerMan, available from the Lawrence Livermore National Laboratory (LLNL), is used to control remote power control (RPC) devices from a central location. PowerMan provides native support for several RPC varieties and expect-like configuration simplifies the addition of new devices.

The latest versions of PowerMan are available at:

sourceforge.net/projects/powerman

For more information about PowerMan, go to:

computing.llnl.gov/linux/powerman.html

8.3.1.2 Power Equipment

Lustre failover also requires the use of RPC devices, which come in different configurations. Lustre server nodes may be equipped with some kind of service processor that allows remote power control. If a Lustre server node is not equipped with a service processor, then a multi-port, Ethernet-addressable RPC may be used as an alternative. For recommended products, refer to the list of supported RPC devices on the PowerMan website.

computing.llnl.gov/linux/powerman.html

8.3.2 Setting up the Heartbeat Software

Lustre must be combined with high-availability (HA) software to enable a complete Lustre failover solution. Lustre can be used with different HA packages, including Heartbeat, the Linux-HA software.

For current information about Heartbeat, see linux-ha.org/wiki.

The Heartbeat package is one of the core components of the Linux-HA project. Heartbeat is highly-portable and runs on every known Linux platform, as well as FreeBSD and Solaris.

This section describes how to install Heartbeat v2 and configure it with and without STONITH. Because Heartbeat v1 has simpler configuration files, which can be used with both Heartbeat v1 and v2, the configuration examples show how to configure Heartbeat using Heartbeat v1 configuration files.

Heartbeat v2 adds monitoring and supports more complex cluster topologies, and the Heartbeat v2 configuration is stored as an XML file. To support users with Heartbeat v2, this section also includes a procedure to migrate Heartbeat v1 configuration files to v2.

8.3.2.1 Installing Heartbeat

1. **Install Lustre (see** Installing Lustre**).**
2. **Install the Heartbeat packages.**

   Heartbeat v2 requires several packages. This example uses Heartbeat v. 2.1.4. The required Heartbeat packages are, in order:

   - **heartbeat-stonith** -> heartbeat-stonith-2.1.4-1.x86_64.rpm
   - **heartbeat-pils** -> heartbeat-pils-2.1.4-1.x86_64.rpm
   - **heartbeat** -> heartbeat-2.1.4-1.x86_64.rpm

   You can download the Heartbeat packages and guides covering basic setup and testing here:

   www.sun.com/software/products/hpcsoftware/getit.jsp

   Heartbeat packages are available for many Linux distributions. Additionally, Heartbeat has some dependencies on other packages. It is recommended that you use a package manager like yum, yast or aptitude to install the Heartbeat packages and resolve their package dependencies.

8.3.2.2 Configuring Heartbeat

This section describes Heartbeat configuration and provides a worked example to illustrate the configuration steps.

Note – Depending on the particular packaging, Heartbeat files may be located in a different directory or path than indicated in the following procedures.

For remote power control, both OSS nodes are equipped with a service processor (SP). The SPs are accessible over the network via their hostnames. Individual node parameters are listed below.

Parameters	Value	Description
First OSS node		
OSS node	`oss01`	First OSS node in the Lustre file system
OST	`ost01`	First OST in the Lustre file system
block device	`/dev/sda`	Block device for the first OSS node (`oss01`)
mount point	`/mnt/ost1`	Mount point for the oss01 block device (/dev/sda) on the `oss01` node
hostname	`oss01sp`	Hostname for the first OSS node's SP
Second OSS node		
OSS node	`oss02`	Second OSS node in the Lustre file system
OST	`ost02`	Second OST in the Lustre file system
block device	`/dev/sdb`	Block device for the second OSS node (`oss02`)
mount point	`/mnt/ost02`	Mount point for the ost02 block device (/dev/sdb) on the `oss02` node
hostname	`oss02sp`	Hostname for the second OSS node's SP

Configuring Heartbeat without STONITH

Note – This procedure describes Heartbeat configuration using a v1 configuration file, which can be used with both Heartbeat v1 and v2. See (Optional) Migrating a Heartbeat Configuration (v1 to v2) for an optional procedure to convert the v1 configuration file to an XML-formatted v2 configuration file.

Note – Depending on the particular packaging, Heartbeat files may be located in a different directory or path than indicated in the following procedure. For example, they may be located in `/etc/ha.d/` or `/var/lib/heartbeat`.

To configure Heartbeat without STONITH:

1. **Create (or edit) the Heartbeat configuration file, /etc/ha.d/ha.cf.**

   This file must be identical on both nodes.

   In this example configuration (without STONITH configuration), the `/etc/ha.d/ha.cf` file looks like this:

```
# log file settings
# write debug output to /var/log/ha-debug
debugfile /var/log/ha-debug
# write log messages to /var/log/ha-log
logfile /var/log/ha-log
# use syslog to write to logfiles
logfacility local0

# set some time-outs. these values are only recommendations, which
depend e.g. on the OSS load
# send keep-alive packages every 2 seconds
keepalive 2
# wait 90 seconds before declaring a node dead
deadtime 90
# write a warning to the logfile after 30 seconds without an answer
from the failover node
warntime 30
# wait for 120 seconds before declaring a node dead after heartbeat
is brought up
initdead 120

# define communication channels
# use port 12345 to communicate with fail-over node
udpport 12345
# use network interfaces eth0 and ib0 to detect a failed node
bcast eth0 ib0

# Use manual failback
auto_failback off

# node names in this failover-pair. These names must match the
output of `hostname`
node oss01
node oss02
```

2. **Define the resources that will be controlled by Heartbeat by editing the /etc/ha/d/haresources file.**

   This file must be identical on both nodes.

   In this example configuration, the /etc/ha.d/haresources file looks like this:

   ```
   oss01 Filesystem::/dev/sda::/mnt/ost01::lustre
   oss02 Filesystem::/dev/sdb::/mnt/ost02::lustre
   ```

   The resource definition file tells Heartbeat that one file system resource is associated with oss01 and oss02. Each resource is defined on separate lines.

   The file system resource script takes three inputs separated by "::". The first parameter is the device name, the second is the mount point and the third is the file system type.

   Depending on the configuration, a resource can be more complex, e.g., software RAID needs to be assembled before the file system can be mounted. In this case, an haresources file may look like this:

   ```
   oss01 Raid1::/etc/mdadm.conf.oss::/dev/md1
   Filesystem::/dev/md1::/mnt/ost01::lustre
   oss02 Raid1::/etc/mdadm.conf.oss::/dev/md2
   Filesystem::/dev/md2::/mnt/ost02::lustre
   ```

   When a resource group is started by Heartbeat, the resources start from left to right. In this example, the RAID is assembled first, and the file system is mounted second. If the resource group is stopped, then the file system is unmounted first and the RAID is stopped second.

   Other resource scripts can be found in the /etc/ha.d/resource.d/ folder.

3. **Create the /etc/ha.d/authkeys file and fix its permissions.**

   This file must be identical on both nodes.

   In this example configuration, the authkeys file looks like this:

   ```
   auth 1
   1 sha1 PutYourSuperSecretKeyHere
   ```

   Make sure that the permissions for this files are set to 0600, by running `chmod 0600 /etc/ha.d/authkeys` on both nodes.

4. **Test the Heartbeat configuration.**

   Run the following command on both nodes:

   ```
   service heartbeat start
   ```

   Check the log files on both nodes to find any problems and fix them.

   After the initial deadtime interval, you should see the nodes discover each other's state and start the Lustre resources associated with them.

Configuring Heartbeat with STONITH

STONITH automates the process of power control and management. Expect scripts are dependent on the exact set of commands provided by each hardware vendor. As a result, any change in the power control hardware or firmware requires that STONITH be adjusted.

Note – This procedure describes configuring Heartbeat using a v1 configuration file, which can be used with both Heartbeat v1 and v2. See (Optional) Migrating a Heartbeat Configuration (v1 to v2) for an optional procedure to convert the v1 configuration file to an XML-formatted v2 configuration file.

Note – Depending on the particular packaging, Heartbeat files may be located in a different directory or path than indicated in the following procedure. For example, they may be located in `/etc/ha.d/` or `/var/lib/heartbeat`.

The heartbeat-stonith package comes with a number of pre-defined STONITH scripts for different power control hardware. Additionally, Heartbeat can be configured to run an external script. Heartbeat can be configured in two STONITH modes:

- One STONITH command for all nodes found in `ha.cf`:

  ```
  stonith <type> <config file>
  ```

- One STONITH command per-node:

  ```
  stonith_host <hostfrom> <stonith_type> <params...>
  ```

You can use an external script to kill each node, e.g.:

```
stonith_host oss01 external foo /etc/ha.d/reset-nodeB
stonith_host oss02 external foo /etc/ha.d/reset-nodeA
```

To get the proper STONITH syntax, run:

```
$ stonith -L
```

The above command lists supported models.

To list required parameters and specify the configuration filename, run:

```
$ stonith -l -t <model>
```

To attempt a test, run:

```
$ stonith -l -t <model> <fake host name>
```

To test STONITH, use a real hostname. To work with Heartbeat correctly, the external STONITH scripts should take the parameters {start | stop | status} and return 0 or 1.

To add STONITH functionality (using an ipmi service processor) to the configuration example, add the following lines to the /etc/ha.d/ha.cf configuration file:

```
# define how a node can be powered off in case of a failure. more
details below
stonith_host oss01 external/ipmi oss02 oss02sp root changeme lanplus
stonith_host oss02 external/ipmi oss01 oss01sp root changeme lanplus
```

STONITH is only invoked if one of the failover nodes is no longer responding to Heartbeat messages and the cluster does stop resources in an orderly manner. If two cluster nodes can communicate, they usually shut down properly. This means that many tests do not produce a STONITH, for example:

- Calling init 0, shutdown, or reboot on a node will cause no STONITH
- Stopping Heartbeat on a node stops the resources cleanly and fails them over to the other node without invoking STONITH.

8.3.2.3 (Optional) Migrating a Heartbeat Configuration (v1 to v2)

Heartbeat includes a script that enables v1 configuration files to be migrated to v2 XML configuration files. The script reads the v1 configuration files (ha.cf and haresources), and then writes an XML file to STDOUT. The script is

```
$ /usr/lib/heartbeat/haresources2cib.py
```

or

```
$ /usr/lib64/heartbeat/haresources2cib.py
```

To redirect the script output after the cib.xml file has been generated, it is recommended that you check the XML file and change some parameters, such as resource-stickiness and timeouts, to more appropriate values. For example:

$ /usr/lib64/heartbeat/haresources2cib.py > cib.xml

Then the cib.xml file should than be copied to /var/lib/heartbeat/crm/cib.xml on both failover nodes.

To test the new configuration, start Heartbeat on both nodes and check the log files.

Note – If a Heartbeat v2 configuration file is available on the system, it is not necessary to remove the v1 configuration files, as they are ignored.

8.3.3 Working with Heartbeat

After Lustre and Heartbeat are correctly configured, the following commands can be used to control Heartbeat.

8.3.3.1 Starting Heartbeat

To start Heartbeat, run this command on both failover nodes:

```
service heartbeat start
```

After a node fails, start Heartbeat manually and analyze the cause of the problem before taking over the failed resources. You should NOT start Heartbeat automatically after a node failure.

8.3.3.2 Switching Resources Between Nodes

Depending on whether Heartbeat v1 or v2 configuration files are being used, there are different ways to switch resources between nodes.

For Heartbeat v1 configuration files, two scripts are provided (`hb_takeover` and `hb_standby`), that make it easy to switch resources between failover nodes. Depending on your system, these scripts are located in `/usr/lib/heartbeat/` or `/usr/lib64/heartbeat/`.

The `hb_takeover` and `hb_standby` scripts take the following arguments:

- **all** -- take/fail over all resources
- **foreign** -- take/fail over foreign resources
- **local** -- take/fail over local resources only
- **failback** -- fail/take over foreign resources

Performing an `hb_takeover` on the current node is equivalent to performing an `hb_standby` on the other node.

For Heartbeat v2 configuration files, the `crm_resource` command is used to interact with Heartbeat's Cluster Resource Manager and switch resources between nodes. For more information on crm_resource, see:

linux.die.net/man/8/crm_resource

To switch resources between nodes:

1. **Generate a complete list of resources known to the Heartbeat cluster resource manager. Run:**

   ```
   crm_resource --list
   ```

2. **From the list, identify the group name for the resource to fail over.**

3. **Determine if and where the specified resource is running. Run:**

   ```
   crm_resource -W -r <resource_name>
   ```

4. **Migrate the resource to the host. Run:**

   ```
   crm_resource -M -r <resource_name> -H <target_host_name>
   ```

5. **To un-migrate a resource, run:**

   ```
   crm_resource -U -r <resource_name>
   ```

CHAPTER 9

Configuring Quotas

This chapter describes how to configure quotas and includes the following sections:

9.1 Working with Quotas

Quotas allow a system administrator to limit the amount of disk space a user or group can use in a directory. Quotas are set by root, and can be specified for individual users and/or groups. Before a file is written to a partition where quotas are set, the quota of the creator's group is checked. If a quota exists, then the file size counts towards the group's quota. If no quota exists, then the owner's user quota is checked before the file is written. Similarly, inode usage for specific functions can be controlled if a user over-uses the allocated space.

Lustre quota enforcement differs from standard Linux quota support in several ways:

- Quotas are administered via the `lfs` command (post-mount).
- Quotas are distributed (as Lustre is a distributed file system), which has several ramifications.
- Quotas are allocated and consumed in a quantized fashion.
- Client does not set the `usrquota` or `grpquota` options to mount. When quota is enabled, it is enabled for all clients of the file system; started automatically using `quota_type` or started manually with `lfs quotaon`.

Caution – Although quotas are available in Lustre, root quotas are NOT enforced.

`lfs setquota -u root` (limits are not enforced)

`lfs quota -u root` (usage includes internal Lustre data that is dynamic in size and does not accurately reflect mount point visible block and inode usage).

9.1.1 Enabling Disk Quotas

Use this procedure to enable (configure) disk quotas in Lustre.

1. **If you have re-complied your Linux kernel, be sure that CONFIG_QUOTA and CONFIG_QUOTACTL are enabled. Also, verify that CONFIG_QFMT_V1 and/or CONFIG_QFMT_V2 are enabled.**

   Quota is enabled in all Linux 2.6 kernels supplied for Lustre.

2. **Start the server.**

3. **Mount the Lustre file system on the client and verify that the lquota module has loaded properly by using the `lsmod` command.**

```
$ lsmod
[root@oss161 ~]# lsmod
Module            Size      Used by
obdfilter         220532    1
fsfilt_ldiskfs    52228     1
ost               96712     1
mgc               60384     1
ldiskfs           186896    2 fsfilt_ldiskfs
lustre            401744    0
lov               289064    1 lustre
lquota            107048    4 obdfilter
mdc               95016     1 lustre
ksocklnd          111812    1
```

The Lustre `mount` command no longer recognizes the `usrquota` and `grpquota` options. If they were previously specified, remove them from `/etc/fstab`.

When quota is enabled, it is enabled for all file system clients (started automatically using `quota_type` or manually with `lfs quotaon`).

Note – Lustre with the Linux kernel 2.4 does not support quotas.

To enable quotas automatically when the file system is started, you must set the `mdt.quota_type` and `ost.quota_type` parameters, respectively, on the MDT and OSTs. The parameters can be set to the string `u` (user), `g` (group) or `ug` for both users and groups.

You can enable quotas at mkfs time (`mkfs.lustre --param mdt.quota_type=ug`) or with tunefs.lustre. As an example:

```
tunefs.lustre --param ost.quota_type=ug $ost_dev
```

Caution – If you are using `mkfs.lustre --param mdt.quota_type=ug` or `tunefs.lustre --param ost.quota_type=ug`, be sure to run the command on all OSTs and the MDT. Otherwise, abnormal results may occur.

9.1.1.1 Administrative and Operational Quotas

Lustre has two kinds of quota files:

- Administrative quotas (for the MDT), which contain limits for users/groups for the entire cluster.
- Operational quotas (for the MDT and OSTs), which contain quota information dedicated to a cluster node.

Lustre 1.6.5 introduced the v2 file format for administrative quota files, with continued support for the old file format (v1). The `mdt.quota_type` parameter also handles '1' and '2' options, to specify the Lustre quota versions that will be used. For example:

```
--param mdt.quota_type=ug1
--param mdt.quota_type=u2
```

Lustre 1.6.6 introduced the v2 file format for operational quotas, with continued support for the old file format (v1). The `ost.quota_type` parameter handles '1' and '2' options, to specify the Lustre quota versions that will be used. For example:

```
--param ost.quota_type=ug2
--param ost.quota_type=u1
```

For more information about the v1 and v2 formats, see Quota File Formats.

9.1.2 Creating Quota Files and Quota Administration

Once each quota-enabled file system is remounted, it is capable of working with disk quotas. However, the file system is not yet ready to support quotas. If umount has been done regularly, run the `lfs` command with the `quotaon` option. If umount has not been done, perform these steps:

1. **Take Lustre "offline". That is, verify that no write operations (append, write, truncate, create or delete) are being performed (preparing to run `lfs quotacheck`). Operations that do not change Lustre files (such as read or mount) are okay to run.**

Caution – When `lfs quotacheck` is run, Lustre must NOT be performing any write operations. Failure to follow this caution may cause the statistic information of quota to be inaccurate. For example, the number of blocks used by OSTs for users or groups will be inaccurate, which can cause unexpected quota problems.

---

2. **Run the `lfs` command with the `quotacheck` option:**

   ```
   # lfs quotacheck -ug /mnt/lustre
   ```

   By default, quota is turned on after `quotacheck` completes. Available options are:

   - `u` — checks the user disk quota information
   - `g` — checks the group disk quota information

   The `lfs quotacheck` command checks all objects on all OSTs and the MDS to sum up for every UID/GID. It reads all Lustre metadata and re-computes the number of blocks/inodes that each UID/GID has used. If there are many files in Lustre, it may take a long time to complete.

Note – User and group quotas are separate. If either quota limit is reached, a process with the corresponding UID/GID cannot allocate more space on the file system.

Note – When `lfs quotacheck` runs, it creates a quota file -- a sparse file with a size proportional to the highest UID in use and UID/GID distribution. As a general rule, if the highest UID in use is large, then the sparse file will be large, which may affect functions such as creating a snapshot.

Note – For Lustre 1.6 releases before version 1.6.5, and 1.4 releases before version 1.4.12, if the underlying ldiskfs file system has not unmounted gracefully (due to a crash, for example), re-run `quotacheck` to obtain accurate quota information. Lustre 1.6.5 and 1.4.12 use journaled quota, so it is not necessary to run `quotacheck` after an unclean shutdown.

In certain failure situations (e.g., when a broken Lustre installation or build is used), re-run `quotacheck` after checking the server kernel logs and fixing the root problem.

The `lfs` command includes several command options to work with quotas:

- `quotaon` — enables disk quotas on the specified file system. The file system quota files must be present in the root directory of the file system.
- `quotaoff` — disables disk quotas on the specified file system.
- `quota` — displays general quota information (disk usage and limits)
- `setquota` — specifies quota limits and tunes the grace period. By default, the grace period is one week.

Usage:

```
lfs quotaon [-ugf] <filesystem>

lfs quotaoff [-ug] <filesystem>

lfs quota [-v] [-o obd_uuid] [-u <username>|-g  <groupname>]
<filesystem>

lfs quota -t <-u|-g> <filesystem>

lfs setquota <-u|--user|-g|--group> <username|groupname>
[-b <block-softlimit>] [-B <block-hardlimit>] [-i <inode-softlimit>]
[-I <inode-hardlimit>] <filesystem>
```

Examples:

In all of the examples below, the file system is /mnt lustre.

To turn on user and group quotas, run:

```
$ lfs quotaon -ug /mnt/lustre
```

To turn off user and group quotas, run:

```
$ lfs quotaoff -ug /mnt/lustre
```

To display general quota information (disk usage and limits) for the user running the command and his primary group, run:

```
$ lfs quota /mnt/lustre
```

To display general quota information for a specific user ("bob" in this example), run:

```
$ lfs quota -u bob /mnt/lustre
```

To display general quota information for a specific user ("bob" in this example) and detailed quota statistics for each MDT and OST, run:

```
$ lfs quota -u bob -v /mnt/lustre
```

To display general quota information for the group to which a specific user ("bob" in this example) belongs, run:

```
$ lfs quota -g bob /mnt/lustre
```

To display general quota information for a specific group ("eng" in this example), run:

```
$ lfs quota -g eng /mnt/lustre
```

To display block and inode grace times for user quotas, run:

```
$ lfs quota -t -u /mnt/lustre
```

To set user and group quotas for a specific user ("bob" in this example), run:

```
$ lfs setquota -u bob 307200 309200 10000 11000 /mnt/lustre
```

In this example, the quota for user "bob" is set to 300 MB (309200*1024) and the hard limit is 11,000 files. Therefore, the inode hard limit should be 11000.

Note – For the Lustre command `$ lfs setquota/quota` ... the qunit for block is KB (1024) and the qunit for inode is 1.

The `quota` command displays the quota allocated and consumed for each Lustre device. Using the previous `setquota` example, running this lfs `quota` command:

```
$ lfs quota -u bob -v /mnt/lustre
```

displays this command output:

```
Disk quotas for user bob (uid 500):

Filesystem   blocks quota  limit  grace  files  quota  limit  grace
/mnt/lustre  0      307200 309200        0      10000  11000
lustre-MDT0000_UUID 0   0  102400        0      0      5000
lustre-OST0000_UUID 0   0  102400
lustre-OST0001_UUID 0   0  102400
```

9.1.3 Quota Allocation

The Linux kernel sets a default quota size of 1 MB. (For a block, the default is 128 MB. For files, the default is 5120.) Lustre handles quota allocation in a different manner. A quota must be properly set or users may experience unnecessary failures. The file system block quota is divided up among the OSTs within the file system. Each OST requests an allocation which is increased up to the quota limit. The quota allocation is then *quantized* to reduce the number of quota-related request traffic. By default, Lustre supports both user and group quotas to limit disk usage and file counts.

The quota system in Lustre is completely compatible with the quota systems used on other file systems. The Lustre quota system distributes quotas from the quota master. Generally, the MDS is the quota master for both inodes and blocks. All OSTs and the MDS are quota slaves to the OSS nodes. The minimum transfer unit is 100 MB, to avoid performance impacts for quota adjustments. The file system block quota is divided up among the OSTs and the MDS within the file system. Only the MDS uses the file system inode quota.

This means that the minimum quota for block is 100 MB* (the number of OSTs + the number of MDSs), which is 100 MB* (number of OSTs + 1). The minimum quota for inode is the inode qunit. If you attempt to assign a smaller quota, users maybe not be able to create files. The default is established at file system creation time, but can be tuned via /proc values (described below). The inode quota is also allocated in a quantized manner on the MDS.

This sets a much smaller granularity. It is specified to request a new quota in units of 100 MB and 500 inodes, respectively. If we look at the setquota example again, running this `lfs quota` command:

```
# lfs quota -u bob -v /mnt/lustre
```

displays this command output:

```
Disk quotas for user bob (uid 500):
Filesystem   blocks quota  limit  grace  files  quota  limit  grace
/mnt/lustre  207432 307200 30920         1041   10000  11000
lustre-MDT0000_UUID 992    0             102400        1041   05000
lustre-OST0000_UUID        103204*             0      102400
lustre-OST0001_UUID        103236*             0      102400
```

The total quota of 30,920 is allotted to user `bob`, which is further distributed to two OSTs and one MDS with a 102,400 block quota.

Note – Values appended with "*" show the limit that has been over-used (exceeding the quota), and receives this message `Disk quota exceeded`. For example:
\
`$ cp: writing `/mnt/lustre/var/cache/fontconfig/beeeeb3dfe132a8a0633a017c99ce0-x86.cache': Disk quota exceeded.`

The requested quota of 300 MB is divided across the OSTs. Each OST has an initial allocation of 100 MB blocks, with iunit limiting to 5000.

Note – It is very important to note that the **block quota is consumed per OST and the MDS per block and inode** (there is only one MDS for inodes). Therefore, when the quota is consumed on one OST, the client may not be able to create files regardless of the quota available on other OSTs.

Additional information:

Grace period — The period of time (in seconds) within which users are allowed to exceed their soft limit. There are four types of grace periods:

- user block soft limit
- user inode soft limit
- group block soft limit
- group inode soft limit

The grace periods are applied to all users. The user block soft limit is for all users who are using a blocks quota.

Soft limit — Once you are beyond the soft limit, the quota module begins to time, but you still can write block and inode. When you are always beyond the soft limit and use up your grace time, you get the same result as the hard limit. For inodes and blocks, it is the same. Usually, the soft limit MUST be less than the hard limit; if not, the quota module never triggers the timing. If the soft limit is not needed, leave it as zero (0).

Hard limit — When you are beyond the hard limit, you get -EQUOTA and cannot write inode/block any more. The hard limit is the absolute limit. When a grace period is set, you can exceed the soft limit within the grace period if are under the hard limits.

Lustre quota allocation is controlled by two values `quota_bunit_sz` and `quota_iunit_sz` referring to KBs and inodes respectively. These values can be accessed on the MDS as `/proc/fs/lustre/mds/*/quota_*` and on the OST as `/proc/fs/lustre/obdfilter/*/quota_*`. The `/proc` values are bounded by two other variables `quota_btune_sz` and `quota_itune_sz`. By default, the `*tune_sz` variables are set at 1/2 the `*unit_sz` variables, and you cannot set `*tune_sz` larger than `*unit_sz`. You must set `bunit_sz` first if it is increasing by more than 2x, and `btune_sz` first if it is decreasing by more than 2x.

Total number of inodes — To determine the total number of inodes, use `lfs df -i` (and also `/proc/fs/lustre/*/*/filestotal`). For more information on using the `lfs df -i` command and the command output, see Querying File System Space.

Unfortunately, the statfs interface does not report the free inode count directly, but instead reports the total inode and used inode counts. The free inode count is calculated for `df` from (total inodes - used inodes).

It is not critical to know a file system's total inode count. Instead, you should know (accurately), the free inode count and the used inode count for a file system. Lustre manipulates the total inode count in order to accurately report the other two values.

The values set for the MDS must match the values set on the OSTs.

The quota_bunit_sz parameter displays bytes, however lfs setquota uses KBs. The quota_bunit_sz parameter must be a multiple of 1024. A proper minimum KB size for lfs setquota can be calculated as:

Size in KBs = (quota_bunit_sz * (number of OSTS + 1)) / 1024

We add one (1) to the number of OSTs as the MDS also consumes KBs. As inodes are only consumed on the MDS, the minimum inode size for lfs setquota is equal to quota_iunit_sz.

Note – Setting the quota below this limit may prevent the user from all file creation.

9.1.4 Known Issues with Quotas

Using quotas in Lustre can be complex and there are several known issues.

9.1.4.1 Granted Cache and Quota Limits

In Lustre, granted cache does not respect quota limits. In this situation, OSTs grant cache to Lustre client to accelerate I/O. Granting cache causes writes to be successful in OSTs, even if they exceed the quota limits, and will overwrite them.

The sequence is:

1. **A user writes files to Lustre.**
2. **If the Lustre client has enough granted cache, then it returns 'success' to users and arranges the writes to the OSTs.**
3. **Because Lustre clients have delivered success to users, the OSTs cannot fail these writes.**

Because of granted cache, writes always overwrite quota limitations. For example, if you set a 400 GB quota on user A and use IOR to write for user A from a bundle of clients, you will write much more data than 400 GB, and cause an out-of-quota error (-EDQUOT).

Note – The effect of granted cache on quota limits can be mitigated, but not eradicated. Reduce the `max_dirty_buffer` in the clients (can be set from 0 to 512). To set `max_dirty_buffer` to 0:

* In releases after Lustre 1.6.5, `lctl set_param osc.*.max_dirty_mb=0`.

* In releases before Lustre 1.6.5, `proc/fs/lustre/osc/*/max_dirty_mb; do echo 512 > $O`

9.1.4.2 Quota Limits

Available quota limits depend on the Lustre version you are using.

- Lustre version 1.4.11 and earlier (for 1.4.x releases) and Lustre version 1.6.4 and earlier (for 1.6.x releases) support quota limits less than 4 TB.
- Lustre versions 1.4.12, 1.6.5 and later support quota limits of 4 TB and greater in Lustre configurations with OST storage limits of 4 TB and less.
- Future Lustre versions are expected to support quota limits of 4 TB and greater with no OST storage limits.

Lustre Version	Quota Limit Per User/Per Group	OST Storage Limit
1.4.11 and earlier	< 4TB	n/a
1.4.12	=> 4TB	<= 4TB of storage
1.6.4 and earlier	< 4TB	n/a
1.6.5	=> 4TB	<= 4TB of storage
Future Lustre versions	=> 4TB	No storage limit

9.1.4.3 Quota File Formats

Lustre 1.6.5 introduced the v2 file format for administrative quotas, with 64-bit limits that support large-limits handling. The old quota file format (v1), with 32-bit limits, is also supported. Lustre 1.6.6 introduced the v2 file format for operational quotas. A few notes regarding the current quota file formats:

Lustre 1.6.5 and later use `mdt.quota_type` to force a specific administrative quota version (v2 or v1).

- For the v2 quota file format, (OBJECTS/admin_quotafile_v2.{usr,grp})
- For the v1 quota file format, (OBJECTS/admin_quotafile.{usr,grp})

Lustre 1.6.6 and later use `ost.quota_type` to force a specific operational quota version (v2 or v1).

- For the v2 quota file format, (lquota_v2.{user,group})
- For the v1 quota file format, (lquota.{user,group})

The `quota_type` specifier can be used to set different combinations of administrative/operational quota file versions on a Lustre node:

- "1" - v1 (32-bit) administrative quota file, v1 (32-bit) operational quota file (default in releases before Lustre 1.6.5)
- "2" - v2 (64-bit) administrative quota file, v1 (32-bit) operational quota file (default in Lustre 1.6.5)
- "3" - v2 (64-bit) administrative quota file, v2 (64-bit) operational quota file (default in releases after Lustre 1.6.5)

If quotas do not exist or look broken, then `quotacheck` creates quota files of a required name and format.

If Lustre is using the v2 quota file format when only v1 quota files exist, then `quotacheck` converts old v1 quota files to new v2 quota files. This conversion is triggered automatically, and is transparent to users. If an old quota file does not exist or looks broken, then the new v2 quota file will be empty. In case of an error, details can be found in the kernel log of the corresponding MDS/OST. During conversion of a v1 quota file to a v2 quota file, the v2 quota file is marked as broken, to avoid it being used if a crash occurs. The quota module does not use broken quota files (keeping quota off).

In most situations, Lustre administrators do not need to set specific versioning options. Upgrading Lustre without using `quota_type` to force specific quota file versions results in quota files being upgraded automatically to the latest version. The option ensures backward compatibility, preventing a quota file upgrade to a version which is not supported by earlier Lustre versions.

9.1.5 Lustre Quota Statistics

Lustre includes statistics that monitor quota activity, such as the kinds of quota RPCs sent during a specific period, the average time to complete the RPCs, etc. These statistics are useful to measure performance of a Lustre file system.

Each quota statistic consists of a quota event and `min_time`, `max_time` and `sum_time` values for the event.

Quota Event	Description
sync_acq_req	Quota slaves send a acquiring_quota request and wait for its return.
sync_rel_req	Quota slaves send a releasing_quota request and wait for its return.
async_acq_req	Quota slaves send an acquiring_quota request and do not wait for its return.
async_rel_req	Quota slaves send a releasing_quota request and do not wait for its return.
wait_for_blk_quota (lquota_chkquota)	Before data is written to OSTs, the OSTs check if the remaining block quota is sufficient. This is done in the lquota_chkquota function.
wait_for_ino_quota (lquota_chkquota)	Before files are created on the MDS, the MDS checks if the remaining inode quota is sufficient. This is done in the lquota_chkquota function.
wait_for_blk_quota (lquota_pending_commit)	After blocks are written to OSTs, relative quota information is updated. This is done in the lquota_pending_commit function.
wait_for_ino_quota (lquota_pending_commit)	After files are created, relative quota information is updated. This is done in the lquota_pending_commit function.
wait_for_pending_blk_quota_req (qctxt_wait_pending_dqacq)	On the MDS or OSTs, there is one thread sending a quota request for a specific UID/GID for block quota at any time. At that time, if other threads need to do this too, they should wait. This is done in the qctxt_wait_pending_dqacq function.
wait_for_pending_ino_quota_req (qctxt_wait_pending_dqacq)	On the MDS, there is one thread sending a quota request for a specific UID/GID for inode quota at any time. If other threads need to do this too, they should wait. This is done in the qctxt_wait_pending_dqacq function.

Quota Event	Description
nowait_for_pending_blk_quota_req (qctxt_wait_pending_dqacq)	On the MDS or OSTs, there is one thread sending a quota request for a specific UID/GID for block quota at any time. When threads enter qctxt_wait_pending_dqacq, they do not need to wait. This is done in the qctxt_wait_pending_dqacq function.
nowait_for_pending_ino_quota_req (qctxt_wait_pending_dqacq)	On the MDS, there is one thread sending a quota request for a specific UID/GID for inode quota at any time. When threads enter qctxt_wait_pending_dqacq, they do not need to wait. This is done in the qctxt_wait_pending_dqacq function.
quota_ctl	The quota_ctl statistic is generated when `lfs setquota`, `lfs quota` and so on, are issued.
adjust_qunit	Each time qunit is adjusted, it is counted.

9.1.5.1 Interpreting Quota Statistics

Quota statistics are an important measure of a Lustre file system's performance. Interpreting these statistics correctly can help you diagnose problems with quotas, and may indicate adjustments to improve system performance.

For example, if you run this command on the OSTs:

```
cat /proc/fs/lustre/lquota/lustre-OST0000/stats
```

You will get a result similar to this:

```
snapshot_time          1219908615.506895 secs.usecs
async_acq_req          1 samples [us]32 32 32
async_rel_req          1 samples [us]5 5 5
nowait_for_pending_blk_quota_req(qctxt_wait_pending_dqacq) 1 samples [us] 2 2 2
quota_ctl              4 samples [us]80 3470 4293
adjust_qunit           1 samples [us]70 70 70
....
```

In the first line, `snapshot_time` indicates when the statistics were taken. The remaining lines list the quota events and their associated data.

In the second line, the `async_acq_req` event occurs one time. The `min_time`, `max_time` and `sum_time` statistics for this event are 32, 32 and 32, respectively. The unit is microseconds (µs).

In the fifth line, the `quota_ctl` event occurs four times. The `min_time`, `max_time` and `sum_time` statistics for this event are 80, 3470 and 4293, respectively. The unit is microseconds (µs).

Involving Lustre Support in Quotas Analysis

Quota statistics are collected in `/proc/fs/lustre/lquota/.../stats`. Each MDT and OST has one statistics proc file. If you have a problem with quotas, but cannot successfully diagnose the issue, send the statistics files in the folder to Lustre Support for analysis. To prepare the files:

1. **Initialize the statistics data to 0 (zero). Run:**

   ```
   lctl set_param lquota.${FSNAME}-MDT*.stats=0
   lctl set_param lquota.${FSNAME}-OST*.stats=0
   ```

2. **Perform the quota operation that causes the problem or degraded performance.**

3. **Collect all statistics in /proc/fs/lustre/lquota/ and send them to Lustre Support. Note the following:**

   - Proc quota entries are collected in these folders:

     `/proc/fs/lustre/obdfilter/lustre-OSTXXXX/quota*`

     - AND -

     `/proc/fs/lustre/mds/lustre-MDTXXXX/quota*`

     Proc quota entries are copied to `/proc/fs/lustre/lquota`.

   - To maintain compatibility, old quota proc entries in the following folders are not deleted in the current Lustre release (although they may be deprecated in the future):

     `/proc/fs/lustre/obdfilter/lustre-OSTXXXX/`

     - AND -

     `/proc/fs/lustre/mds/lustre-MDTXXXX/`

   - Only use the quota entries in `/proc/fs/lustre/lquota/`.

CHAPTER 10

RAID

This chapter describes software and hardware RAID, and includes the following sections:

10.1 Considerations for Backend Storage

Lustre's architecture allows it to use any kind of block device as backend storage. The characteristics of such devices, particularly in the case of failures vary significantly and have an impact on configuration choices.

This section surveys issues and recommendations regarding backend storage.

10.1.1 Selecting Storage for the MDS or OSTs

MDS

The MDS does a large amount of small writes. For this reason, we recommend that you use RAID1 for MDT storage. If you require more capacity for an MDT than one disk provides, we recommend RAID1 + 0 or RAID10. LVM is not recommended at this time for performance reasons.

OST

A quick calculation (shown below), makes it clear that without further redundancy, RAID5 is not acceptable for large clusters and RAID6 is a must.

Take a 1 PB file system (2,000 disks of 500 GB capacity). The MTTF[1] of a disk is about 1,000 days. This means that the expected failure rate is 2000/1000 = 2 disks per day. Repair time at 10% of disk bandwidth is close to 1 day (500 GB at 5 MB/sec = 100,000 sec = 1 day).

If we have a RAID 5 stripe that is 10 disks wide, then during 1 day of rebuilding, the chance that a second disk in the same array fails is about 9 / 1000 ~= 1/100. This means that, in the expected period of 50 days, a double failure in a RAID 5 stripe leads to data loss.

So, RAID 6 or another double parity algorithm is necessary for OST storage.

For better performance, we recommend that you create RAID sets with no more than 8 data disks (+1 or +2 parity disks) as this will provide more IOPS from having multiple independent RAID sets.

1. Mean Time to Failure

File system: Use RAID5 with 5 or 9 disks or RAID6 with 6 or 10 disks, each on a different controller. The stripe width is the optimal minimum I/O size. Ideally, the RAID configuration should allow 1 MB Lustre RPCs to fit evenly on a single RAID stripe without an expensive read-modify-write cycle. Use this formula to determine the stripe_width.

```
<stripe_width> = <chunksize> * (<disks> - <parity_disks>) <= 1 MB
```

where <parity_disks> is 1 for RAID5/RAID-Z and 2 for RAID6/RAID-Z2. If the RAID configuration does not allow <chunksize> to fit evenly into 1 MB, select <chunksize>, such that <stripe_width> is close to 1 MB, but not larger.

For example, RAID6 with 6 disks has 4 data and 2 parity disks, so we get:

<chunksize> <= 1024kB/4; either 256kB, 128kB or 64kB

The <stripe_width> value must equal <chunksize> * (<disks> - <parity_disks>). Use it for OST file systems only (not MDT file systems).

```
$ mkfs.lustre --mountfsoptions="stripe=<stripe_width_blocks>" ...
```

External journal: Use RAID1 with two partitions of 400 MB (or more), each from disks on different controllers.

To set up the journal device (/dev/mdJ), run:

```
$ 'mke2fs -O journal_dev -b 4096 /dev/mdJ'
```

Then run --reformat on the file system device (/dev/mdX), specifying the RAID geometry to the underlying ldiskfs file system, where:

```
<chunk_blocks> = <chunksize> / 4096
<stripe_width_blocks> = <stripe_width> / 4096:

$ mkfs.lustre --reformat ...
--mkfsoptions "-j -J device=/dev/mdJ -E stride=<chunk_blocks>" /dev/mdX
```

10.1.2 Reliability Best Practices

It is considered mandatory that you use disk monitoring software, so rebuilds happen without any delay.

We recommend backups of the metadata file systems. This can be done with LVM snapshots or using raw partition backups.

10.1.3 Understanding Double Failures with Hardware and Software RAID5

Software RAID does not offer the hard consistency guarantees of top-end enterprise RAID arrays. Hardware RAID guarantees that the value of any block is exactly the before or after value and that ordering of writes is preserved. With software RAID, an interrupted write operation that spans multiple blocks can frequently leave a stripe in an inconsistent state that is not restored to either the old or the new value. Normally, such interruptions are caused by an abrupt shutdown of the system.

If the array functions without disk failures, but experiences sudden power-down incidents, such as interrupted writes on journal file systems, these events can affect file data and data in the journal. Metadata itself is re-written from the journal during recovery and is correct. Because the journal uses a single block to indicate a complete transaction has committed after other journal writes have completed, the journal remains valid. File data can be corrupted when overwriting file data; this is a known problem with incomplete writes and caches. Recovery of the disk file systems with software RAID is similar to recovery without software RAID. Using Lustre servers with disk file systems does not change these guarantees.

Problems can arise if, after an abrupt shutdown, a disk fails on restart. In this case, even single block writes provide no guarantee that (as an example), the journal will not be corrupted. Follow these requirements:

- If the power down of a system using software RAID is followed by a disk failure before the RAID array can be re-synchronized, the disk file system needs a file system check and any data that was being written during the power loss may be corrupted.
- If a RAID array does not guarantee before/after semantics, the same requirement holds.

  We consider this to be a requirement for most arrays that are used with Lustre, including the successful and popular DDN arrays.

  With RAID6 this check is not required with a single disk failure, but is required with a double failure upon reboot after an abrupt interruption of the system.

10.1.4 Performance Tradeoffs

Writeback cache can dramatically increase write performance on any type of RAID array.[2] Unfortunately, unless the RAID array has battery-backed cache (a feature only found in some higher-priced hardware RAID arrays), interrupting the power to the array may result in out-of-sequence writes. This causes problems for journaling.

If writeback cache is enabled, a file system check is required after the array loses power. Data may also be lost because of this.

Therefore, we recommend against the use of writeback cache when data integrity is critical. You should carefully consider whether the benefits of using writeback cache outweigh the risks.

10.1.5 Formatting Options for RAID Devices

When formatting a file system on a RAID device, it is beneficial to specify additional parameters at the time of formatting. This ensures that the file system is optimized for the underlying disk geometry. Use the `--mkfsoptions` parameter to specify these options when formatting the OST or MDT.

For RAID 5, RAID 6, RAID 1+0 storage, specifying the `-E stride = <chunksize>` option improves the layout of the file system metadata ensuring that no single disk contains all of the allocation bitmaps. The <chunksize> parameter is in units of 4096-byte blocks and represents the amount of contiguous data written to a single disk before moving to the next disk. This is applicable to both MDS and OST file systems.

For more information on how to override the defaults while formatting MDS or OST file systems, see Options for Formatting the MDT and OSTs.

2. Client writeback cache improves performance for many small files or for a single, large file alike. However, if the cache is filled with small files, cache flushing is likely to be much slower (because of less data being sent per RPC), so there may be a drop-off in total throughput.

10.1.5.1 Creating an External Journal

If you have configured a RAID array and use it directly as an OST, it houses both data and metadata. For better performance[3], we recommend putting OST metadata on another journal device, by creating a small RAID 1 array and using it as an external journal for the OST.

It is not known if external journals improve performance of MDTs. Currently, we recommend against using them for MDTs to reduce complexity.

No more than 102,400 file system blocks will ever be used for a journal. For Lustre's standard 4 KB block size, this corresponds to a 400 MB journal. A larger partition can be created, but only the first 400 MB will be used. Additionally, a copy of the journal is kept in RAM on the OSS. Therefore, make sure you have enough memory available to hold copies of all the journals.

To create an external journal, perform these steps for each OST on the OSS:

1. **Create a 400 MB (or larger) journal partition (RAID 1 is recommended).**

   In this example, /dev/sdb is a RAID 1 device, run:

   ```
   $ sfdisk -uC /dev/sdb << EOF
   > ,50,L
   > EOF
   ```

2. **Create a journal device on the partition. Run:**

   ```
   $ mke2fs -b 4096 -O journal_dev /dev/sdb1
   ```

3. **Create the OST.**

   In this example, /dev/sdc is the RAID 6 device to be used as the OST, run:

   ```
   $ mkfs.lustre --ost --mgsnode=mds@osib \
   --mkfsoptions="-J device=/dev/sdb1" /dev/sdc
   ```

4. **Mount the OST as usual.**

3. Performance is affected because, while writing large sequential data, small I/O writes are done to update metadata. This small-sized I/O can affect performance of large sequential I/O with disk seeks.

10.1.6 Handling Degraded RAID Arrays

Lustre 1.8.2 and later versions include functionality that notifies Lustre if an external RAID array has degraded performance (resulting in a degraded OST), either because a disk has failed and not been replaced, or because a disk was replaced and is undergoing a rebuild. To avoid a global performance slowdown due to a degraded OST, the MDS can avoid the OST for new object allocation if it is notified of the degraded state.

The new file (called "`degraded`"), in `/proc/fs/lustre/obdfilter/{OST}`, marks the OST as degraded if it is written with a "1" (or any non-zero value), until a "0" is written to it. Therefore, "1" should be written to the file when the array becomes degraded and "0" should be written when the array becomes healthy.

If the OST is remounted due to a reboot or other condition, the flag resets to "0".

10.2 Insights into Disk Performance Measurement

Several tips and insights for disk performance measurement are provided below. Some of this information is specific to RAID arrays and/or the Linux RAID implementation.

- Performance is limited by the slowest disk.

  Before creating a software RAID array, benchmark all disks individually. We have frequently encountered situations where drive performance was not consistent for all devices in the array. Replace any disks that are significantly slower than the rest.

- Disks and arrays are very sensitive to request size.

  To identify the optimal request size for a given disk, benchmark the disk with different record sizes ranging from 4 KB to 1 to 2 MB.

Note – Try to avoid sync writes; probably subsequent write would make the stripe full and no reads will be needed. Try to configure RAID arrays and the application so that most of the writes are full-stripe and stripe-aligned.

- (Suggested) MDT setup for maximum performance.

  RAID1 with an internal journal and two disks from different controllers.

  If you need a larger MDT, create multiple RAID1 devices from pairs of disks, and then make a RAID0 array of the RAID1 devices. This ensures maximum reliability because multiple disk failures only have a small chance of hitting both disks in the same RAID1 device.

  Doing the opposite (RAID1 of a pair of RAID0 devices) has a 50% chance that even two disk failures can cause the loss of the whole MDT device. The first failure disables an entire half of the mirror and the second failure has a 50% chance of disabling the remaining mirror.

10.3 Lustre Software RAID Support

A number of Linux kernels offer software RAID support, by which the kernel organizes disks into a RAID array. All Lustre-supported kernels have software RAID capability, but Lustre has added performance improvements to the RHEL 4 and RHEL 5 kernels that make operations even faster[4]. Therefore, if you are using software RAID functionality, we recommend that you use a Lustre-patched RHEL 4 or 5 kernel to take advantage of these performance improvements, rather than a SLES kernel.

10.3.0.1 Enabling Software RAID on Lustre

This procedure describes how to set up software RAID on a Lustre system. It requires use of mdadm, a third-party tool to manage devices using software RAID.

1. **Install Lustre, but do not configure it yet. See** Installing Lustre**.**

2. **Create the RAID array with the** `mdadm` **command.**

   The `mdadm` command is used to create and manage software RAID arrays in Linux, as well as to monitor the arrays while they are running. To create a RAID array, use the `--create` option and specify the MD device to create, the array components, and the options appropriate to the array.

Note – For best performance, we generally recommend using disks from as many controllers as possible in one RAID array.

4. These enhancements have mostly improved write performance.

To illustrate how to create a software RAID array for Lustre, the steps below include a worked example that creates a 10-disk RAID 6 array from disks `/dev/dsk/c0t0d0` through `c0tod4` and `/dev/dsk/c1t0d0` through `c1tod4`. This RAID array has no spares.

For the 10-disk RAID 6 array, there are 8 active disks. The chunk size must be chosen such that `<chunksize>` <= 1024KB/8. Therefore, the largest valid chunk size is 128KB.

a. Create a RAID array for an OST. On the OSS, run:

```
$ mdadm --create <array_device> -c <chunksize> -l \
<raid_level> -n <active_disks> -x <spare_disks> <block_devices>
```

where:

<array_device>	RAID array to create, in the form of `/dev/mdX`
<chunksize>	Size of each stripe piece on the array's disks (in KB); discussed above.
<raid_level>	Architecture of the RAID array. RAID 5 and RAID 6 are commonly used for OSTs.
<active_disks>	Number of active disks in the array, including parity disks.
<spare_disks>	Number of spare disks initially assigned to the array. More disks may be brought in via spare pooling (see below).
<block_devices>	List of the block devices used for the RAID array; wildcards may be used.

For the worked example, the command is:

```
$ mdadm --create /dev/md10 -c 128 -l 6 -n 10 -x 0 \
/dev/dsk/c0t0d[01234] /dev/dsk/c1t0d[01234]
```

This command output displays:

```
mdadm: array /dev/md10 started.
```

We also want an external journal on a RAID 1 device. We create this from two 400MB partitions on separate disks: /dev/dsk/c9t0d20p1 and /dev/dsk/c1t0d20p1

b. Create a RAID array for an external journal. On the OSS, run:

```
$ mdadm --create <array_device> -l <raid_level> -n \
<active_devices> -x <spare_devices> <block_devices>
```

where:

`<array_device>`	RAID array to create, in the form of /dev/md*X*
`<raid_level>`	Architecture of the RAID array. RAID 1 is recommended for external journals.
`<active_devices>`	Number of active disks in the RAID array, including mirrors.
`<spare_devices>`	Number of spare disks initially assigned to the RAID array. More disks may be brought in via spare pooling (see below).
`<block_devices>`	List of the block devices used for the RAID array; wildcards may be used.

For the worked example, the command is:

```
$ mdadm --create /dev/md20 -l 1 -n 2 -x 0 /dev/dsk/c0t0d20p1 \
/dev/dsk/c1t0d20p1
```

This command output displays:

```
mdadm: array /dev/md20 started.
```

We now have two arrays - a RAID 6 array for the OST (`/dev/md20`), and a RAID 1 array for the external journal (`/dev/md20`).

The arrays will now be re-synced, a process which re-synchronizes the various disks in the array so their contents match. The arrays may be used during the re-sync process (including formatting the OSTs), but performance will not be as high as usual. The re-sync progress may be monitored by reading the `/proc/mdstat` file.

Next, you need to create a RAID array for an MDT. In this example, a RAID 10 array is created with 4 disks: `/dev/dsk/c0t0d1`, `c0t0d3`, `c1t0d1`, and `c1t0d3`. For smaller arrays, RAID 1 could be used.

c. Create a RAID array for an MDT. On the MDT, run:

```
$ mdadm --create <array_device> -l <raid_level> -n \
<active_devices> -x <spare_devices> <block_devices>
```

where:

`<array_device>`	RAID array to create, in the form of /dev/md*X*
`<raid_level>`	Architecture of the RAID array. RAID 1 or RAID 10 is recommended for MDTs.
`<active_devices>`	Number of active disks in the RAID array, including mirrors.
`<spare_devices>`	Number of spare disks initially assigned to the RAID array. More disks may be brought in via spare pooling (see below).
`<block_devices>`	List of the block devices used for the RAID array; wildcards may be used.

For the worked example, the command is:

```
$ mdadm --create -l 10 -n 4 -x 0 /dev/md10 /dev/dsk/c[01]t0d[13]
```

This command output displays:

```
mdadm: array /dev/md10 started.
```

If you creating many arrays across many servers, we recommend scripting this process.

Note – Do not use the `--assume-clean` option when creating arrays. This could lead to data corruption on RAID 5 and will cause array checks to show errors with all RAID types.

---

3. **Set up the mdadm tool.**

   The mdadm tool enables you to monitor disks for failures (you will receive a notification). It also enables you to manage spare disks. When a disk fails, you can use mdadm to make a spare disk active, until such time as the failed disk is replaced.

   Here is an example mdadm.conf from an OSS with 7 OSTs including external journals. Note how spare groups are configured, so that OSTs without spares still benefit from the spare disks assigned to other OSTs.

```
ARRAY /dev/md10 level=raid6 num-devices=10
    UUID=e8926d28:0724ee29:65147008:b8df0bd1 spare-group=raids
ARRAY /dev/md11 level=raid6 num-devices=10 spares=1
    UUID=7b045948:ac4edfc4:f9d7a279:17b468cd spare-group=raids
ARRAY /dev/md12 level=raid6 num-devices=10 spares=1
    UUID=29d8c0f0:d9408537:39c8053e:bd476268 spare-group=raids
ARRAY /dev/md13 level=raid6 num-devices=10
    UUID=1753fa96:fd83a518:d49fc558:9ae3488c spare-group=raids
ARRAY /dev/md14 level=raid6 num-devices=10 spares=1
    UUID=7f0ad256:0b3459a4:d7366660:cf6c7249 spare-group=raids
ARRAY /dev/md15 level=raid6 num-devices=10
    UUID=09830fd2:1cac8625:182d9290:2b1ccf2a spare-group=raids
ARRAY /dev/md16 level=raid6 num-devices=10
    UUID=32bf1b12:4787d254:29e76bd7:684d7217 spare-group=raids
ARRAY /dev/md20 level=raid1 num-devices=2 spares=1
    UUID=bcfb5f40:7a2ebd50:b3111587:8b393b86 spare-group=journals
ARRAY /dev/md21 level=raid1 num-devices=2 spares=1
    UUID=6c82d034:3f5465ad:11663a04:58fbc2d1 spare-group=journals
ARRAY /dev/md22 level=raid1 num-devices=2 spares=1
    UUID=7c7274c5:8b970569:03c22c87:e7a40e11 spare-group=journals
ARRAY /dev/md23 level=raid1 num-devices=2 spares=1
    UUID=46ecd502:b39cd6d9:dd7e163b:dd9b2620 spare-group=journals
ARRAY /dev/md24 level=raid1 num-devices=2 spares=1
    UUID=5c099970:2a9919e6:28c9b741:3134be7e spare-group=journals
ARRAY /dev/md25 level=raid1 num-devices=2 spares=1
    UUID=b44a56c0:b1893164:4416e0b8:75beabc4 spare-group=journals
ARRAY /dev/md26 level=raid1 num-devices=2 spares=1
    UUID=2adf9d0f:2b7372c5:4e5f483f:3d9a0a25 spare-group=journals

# Email address to notify of events (e.g. disk failures)
MAILADDR admin@example.com
```

4. **Set up periodic checks of the RAID array.**

   We recommend checking the software RAID arrays monthly for consistency. This can be done using cron and should be scheduled for an idle period so performance is not affected.

   To start a check, write "check" into `/sys/block/[ARRAY]/md/sync_action`. For example, to check /dev/md10, run this command on the Lustre server:

   ```
   $ echo check > /sys/block/md10/md/sync_action
   ```

5. **Format the OSTs and MDT, and continue with normal Lustre setup and configuration.**

   For configuration information, see Configuring Lustre.

Note – Per Bugzilla 18475, we recommend that `stripe_cache_size` be set to 16KB (instead of 2KB).

These additional resources may be helpful when enabling software RAID on Lustre:

- md(4), mdadm(8), mdadm.conf(5) manual pages
- Linux software RAID wiki: http://linux-raid.osdl.org/
- Kernel documentation: Documentation/md.txt

CHAPTER 11

Kerberos

This chapter describes how to use Kerberos with Lustre and includes the following sections:

11.1 What is Kerberos?

Kerberos is a mechanism for authenticating all entities (such as users and services) on an “unsafe” network. Users and services, known as "principals", share a secret password (or key) with the Kerberos server. This key enables principals to verify that messages from the Kerberos server are authentic. By trusting the Kerberos server, users and services can authenticate one another.

Caution – Kerberos is a future Lustre feature that is not available in current versions. If you want to test Kerberos with a pre-release version of Lustre, check out the Lustre source from the CVS repository and build it. For more information on checking out Lustre source code, see CVS.

11.2 Lustre Setup with Kerberos

Setting up Lustre with Kerberos can provide advanced security protections for the Lustre network. Broadly, Kerberos offers three types of benefit:

- Allows Lustre connection peers (MDS, OSS and clients) to authenticate one another.
- Protects the integrity of the PTLRPC message from being modified during network transfer.
- Protects the privacy of the PTLRPC message from being eavesdropped during network transfer.

Kerberos uses the “kernel keyring” client upcall mechanism.

11.2.1 Configuring Kerberos for Lustre

This section describes supported Kerberos distributions and how to set up and configure Kerberos on Lustre.

11.2.1.1 Kerberos Distributions Supported on Lustre

Lustre supports the following Kerberos distributions:

- MIT Kerberos 1.3.x
- MIT Kerberos 1.4.x
- MIT Kerberos 1.5.x
- MIT Kerberos 1.6 (not yet verified)

On a number of operating systems, the Kerberos RPMs are installed when the operating system is first installed. To determine if Kerberos RPMs are installed on your OS, run:

```
# rpm -qa | grep krb
```

If Kerberos is installed, the command returns a list like this:

```
krb5-devel-1.4.3-5.1
krb5-libs-1.4.3-5.1
krb5-workstation-1.4.3-5.1
pam_krb5-2.2.6-2.2
```

Note – The Heimdal implementation of Kerberos is not currently supported on Lustre, although it support will be added in an upcoming release.

11.2.1.2 Preparing to Set Up Lustre with Kerberos

To set up Lustre with Kerberos:

1. **Configure NTP to synchronize time across all machines.**
2. **Configure DNS with zones.**
3. **Verify that there are fully-qualified domain names (FQDNs), that are resolvable in both forward and reverse directions for all servers. This is required by Kerberos.**
4. **On every node, install flowing packages:**
   - libgssapi (version 0.10 or higher)

     Some newer Linux distributions include libgssapi by default. If you do not have libgssapi, build and install it from source:
     http://www.citi.umich.edu/projects/nfsv4/linux/libgssapi/libssapi-0.10.tar.gz
   - keyutils

11.2.1.3 Configuring Lustre for Kerberos

To configure Lustre for Kerberos:

1. **Configure the client nodes.**

   a. **For each client node, create a lustre_root principal and generate the keytab.**

   ```
   kadmin> addprinc -randkey lustre_root/client_host.domain@REALM
   kadmin> ktadd -e aes128-cts:normal lustre_root/client_host.domain@REALM
   ```

   b. **Install the keytab on the client node.**

Note – For each client-OST pair, there is only one security context, shared by all users on the client. This protects data written by one user to be passed to an OST by another user due to asynchronous bulk I/O. The client-OST connection only guarantees message integrity or privacy; it does not authenticate users.

---

2. **Configure the MDS nodes.**

   a. **For each MDS node, create a lustre_mds principal and generate the keytab.**

   ```
   kadmin> addprinc -randkey lustre_mds/mdthost.domain@REALM
   kadmin> ktadd -e aes128-cts:normal lustre_mds/mdthost.domain@REALM
   ```

   b. **Install the keytabl on the MDS node.**

3. **Configure the OSS nodes.**

   a. **For each OSS node, create a lustre_oss principal and generate the keytab.**

   ```
   kadmin> addprinc -randkey lustre_oss/osthost.domain@REALM
   kadmin> ktadd -e aes128-cts:normal lustre_oss/osshost.domain@REALM
   ```

   b. **Install the keytab on the OSS node.**

Tip – To avoid assigning a unique keytab to each client node, create a general lustre_root principal and keytab, and install the keytab on as many client nodes as needed.

```
kadmin> addprinc -randkey lustre_root@REALM
kadmin> ktadd -e aes128-cts:normal lustre_root@REALM
```

Remember that if you use a general keytab, then one compromised client means that all client nodes are insecure.

General Installation Notes

- The *host.domain* should be the FQDN in your network. Otherwise, the server may not recognize any GSS request.
- To install a keytab entry on a node, use the **ktutil**[1] utility.
- Lustre supports these encryption types for MIT Kerberos 5, v1.4 and higher:
  - **des-cbc-crc**
  - **des-cbc-md5**
  - **des3-hmac-sha1**
  - **aes128-cts**
  - **aes256-cts**
  - arcfour-hmac-md5

  For MIT Kerberos 1.3.x, only **des-cbc-md5** works because of a known issue between libgssapi and the Kerberos library.

Note – The encryption type (or enctype) is an identifier specifying the encryption, mode and hash algorithms. Each Kerberos key has an associated enctype that identifies the cryptographic algorithm and mode used when performing cryptographic operations with the key. It is important that the enctypes requested by the client are actually supported on the system hosting the client. This is the case if the defaults that control enctypes are not overridden.

---

1. Kerberos keytab file maintenance utility.

11.2.1.4 Configuring Kerberos

To configure Kerberos to work with Lustre:

1. **Modify the files for Kerberos:**

```
$ /etc/krb5.conf
[libdefaults]
default_realm = CLUSTERFS.COM

[realms]
CLUSTERFS.COM = {
kdc = mds16.clustrefs.com
admin_server = mds16.clustrefs.com
}

[domain_realm]
.clustrefs.com = CLUSTERFS.COM
clustrefs.com = CLSUTREFS.COM

[logging]
default = FILE:/var/log/kdc.log
```

2. **Prepare the Kerberos database.**
3. **Create service principals so Lustre supports Kerberos authentication.**

Note – You can create service principals when configuring your other services to support Kerberos authentication.

---

4. **Configure the client nodes. For each client node:**

   a. **Create a lustre_root principal and generate the keytab:**

   ```
   kadmin> addprinc -randkey lustre_root/client_host.domain@REALM
     kadmin> ktadd -e aes128-cts:normal
   lustre_root/client_host.domain@REALM
   ```

   This process populates `/etc/krb5.keytab`, which is not human-readable. Use the `ktutil` program to read and modify it.

   b. **Install the keytab.**

Note – There is only one security context for each client-OST pair, shared by all users on the client. This protects data written by one user to be passed to an OST by another user due to asynchronous bulk I/O. The client-OST connection only guarantees message integrity or privacy; it does not authenticate users.

---

5. **Configure the MDS nodes. For each MDT node, create a lustre_mds principal, and generate and install the keytab.**

   ```
   kadmin> addprinc -randkey lustre_mds/mdthost.domain@REALM
     kadmin> ktadd -e aes128-cts:normal
   lustre_mds/mdthost.domain@REALM
   ```

6. **Configure the OSS nodes. For each OST node, create a lustre_oss principal, and generate and install the keytab.**

   ```
   kadmin> addprinc -randkey lustre_oss/oss_host.domain@REALM
     kadmin> ktadd -e aes128-cts:normal
   lustre_oss/oss_host.domain@REALM
   ```

To save the trouble of assigning a unique keytab for each client node, create a general lustre_root principal and its keytab, and then install the keytab on as many client nodes as needed.

```
kadmin> addprinc -randkey lustre_root@REALM
kadmin> ktadd -e aes128-cts:normal lustre_root@REALM
```

Note – If one client is compromised, all client nodes become insecure.

For more detailed information on installing and configuring Kerberos, see:

http://web.mit.edu/Kerberos/krb5-1.6/#documentation

11.2.1.5 Setting the Environment

Perform the following steps to configure the system and network to use Kerberos.

System-wide Configuration

1. **On each MDT, OST, and client node, add the following line to /etc/fstab to mount them automatically.**

```
nfsd       /proc/fs/nfsd    nfsd   defaults  0   0
```

2. **On each MDT and client node, dd the following line to /etc/request-key.conf.**

```
create lgssc * * /usr/sbin/lgss_keyring %o %k %t %d %c %u %g %T %P %S
```

Networking

If your network is not using SOCKLND or InfiniBand (and uses Quadrics, Elan or Myrinet, for example), configure a /etc/lustre/nid2hostname (simple script that translates a NID to a hostname) on each server node (MDT and OST). This is an example on an Elan cluster:

```
#!/bin/bash
set -x
exec 2>/tmp/$(basename $0).debug

# convert a NID for a LND to a hostname, for GSS for example

# called with three arguments: lnd netid nid
#   $lnd will be string "QSWLND", "GMLND", etc.
#   $netid will be number in hex string format, like "0x16", etc.
#   $nid has the same format as $netid
# output the corresponding hostname, or error message leaded by a '@'
for error logging.

lnd=$1
netid=$2
nid=$3
```

```
# uppercase the hex
nid=$(echo $nid | tr '[abcdef]' '[ABCDEF]')
# and convert to decimal
nid=$(echo -e "ibase=16\n${nid/#0x}" | bc)
case $lnd in

          QSWLND)   # simply stick "mtn" on the front

                    echo "mtn$nid"
                    ;;

                    *)        echo "@unknown LND: $lnd"

                    ;;
esac
```

11.2.1.6 Building Lustre

If you are compiling the kernel from the source, enable GSS during configuration:

```
# ./configure --with-linux=path_to_linux_source --enable-gss - \
other-options
```

When you enable Lustre with GSS, the configuration script checks all dependencies, like Kerberos and `libgssapi` installation, and in-kernel SUNRPC-related facilities. When you install `lustre-xxx.rpm` on target machines, RPM again checks for dependencies like Kerberos and `libgssapi`.

11.2.1.7 Running GSS Daemons

If you turn on GSS between an MDT-OST or MDT-MDT, GSS treats the MDT as a client. You should run `lgssd` on the MDT.

There are two types of GSS daemons: `lgssd` and `lsvcgssd`. Before starting Lustre, make sure they are running on each node:

- OST: `lsvcgssd`
- MDT: `lsvcgssd`
- CLI: none

Note – Verbose logging can help you make sure Kerberos is set up correctly. To use verbose logging and run it in the foreground, run `lsvcgssd -vvv -f`

`-v` increases the verbose level of a debugging message by 1. For example, to set the verbose level to 3, run `lsvcgssd -v -v -v`

`-f` runs `lsvcgssd` in the foreground, instead of as daemon.

We are maintaining a patch against nfs-utils, and bringing necessary patched files into the Lustre tree. After a successful build, GSS daemons are built under `lustre/utils/gss` and are part of `lustre-xxxx.rpm`.

11.2.2 Types of Lustre-Kerberos Flavors

There are three major flavors in which you can configure Lustre with Kerberos:

- Basic Flavors
- Security Flavor
- Customized Flavor

Select a flavor depending on your priorities and preferences.

11.2.2.1 Basic Flavors

Currently, we support six basic flavors: ***null***, ***plain***, ***krb5n***, ***krb5a***, ***krb5i***, and ***krb5p***.

Basic Flavor	Authentication	RPC Message Protection	Bulk Data Protection	Remarks
null	N/A	N/A	N/A*	Almost no performance overhead. The on-wire RPC data is compatible with old versions of Lustre (1.4.x, 1.6.x).
plain	N/A	null	checksum (adler32)	Carries checksum (which only protects data mutating during transfer, cannot guarantee the genuine author because there is no actual authentication).
krb5n	GSS/Kerberos5	null	checksum (adler32)	No protection of the RPC message, adler32 checksum protection of bulk data; light performance overhead.

Basic Flavor	Authentication	RPC Message Protection	Bulk Data Protection	Remarks
krb5a	GSS/Kerberos5	partial integrity	checksum (adler32)	Only the header of the RPC message is integrity protected, adler32 checksum protection of bulk data, more performance overhead compared to **krb5n**.
krb5i	GSS/Kerberos5	integrity	integrity [sha1]	RPC message integrity protection algorithm is determined by actual Kerberos algorithms in use; heavy performance overhead.
krb5p	GSS/Kerberos5	privacy	privacy [sha1/aes128]	RPC message privacy protection algorithm is determined by actual Kerberos algorithms in use; heaviest performance overhead.

* In Lustre 1.6.5, bulk data checksumming is enabled (by default) to provide integrity checking using the adler32 mechanism if the OSTs support it. Adler32 checksums offer lower CPU overhead than CRC32.

11.2.2.2 Security Flavor

A security flavor is a string that describes what kind of security transform is performed on a given PTLRPC connection. It covers two parts of messages, the RPC message and BULK data. You can set either part in one of the following modes:

- ***null*** – No protection
- ***integrity*** – Data integrity protection (checksum or signature)
- ***privacy*** – Data privacy protection (encryption)

11.2.2.3 Customized Flavor

In most situations, you do not need a customized flavor, a basic flavor is sufficient for regular use. But to some extent, you can customize the flavor string. The flavor string format is:

```
base_flavor[-bulk{nip}[:hash_alg[/cipher_alg]]]
```

Here are some examples of customized flavors:

plain-bulkn

Use ***plain*** on the RPC message (null protection), and no protection on the bulk transfer.

krb5i-bulkn

Use ***krb5i*** on the RPC message, but do not protect the bulk transfer.

krb5p-bulki

Use ***krb5p*** on the RPC message, and protect data integrity of the bulk transfer.

krb5p-bulkp:sha512/aes256

Use ***krb5p*** on the RPC message, and protect data privacy of the bulk transfer by algorithm SHA512 and AES256.

Currently, Lustre supports these bulk data cryptographic algorithms:

- Hash:
  - ***adler32***
  - ***crc32***
  - ***md5***
  - ***sha1 / sha256 / sha384 / sha512***
  - ***wp256 / wp384 / wp512***
- Cipher:
  - ***arc4***
  - ***aes128 / aes192 / aes256***
  - ***cast128 / cast256***
  - ***twofish128 / twofish256***

11.2.2.4 Specifying Security Flavors

If you have not specified a security flavor, the CLIENT-MDT connection defaults to ***plain***, and all other connections use null.

Specifying Flavors by Mount Options

When mounting OST or MDT devices, add the mount option (shown below) to specify the security flavor:

```
# mount -t lustre -o sec=plain /dev/sda1 /mnt/mdt/
```

This means all connections to this device will use the plain flavor. You can split this sec=flavor as:

```
# mount -t lustre -o sec_mdt={flavor1},sec_cli={flavor1}/dev/sda \
/mnt/mdt/
```

This means connections from other MDTs to this device will use flavor1, and connections from all clients to this device will use flavor2.

Specifying Flavors by On-Disk Parameters

You can also specify the security flavors by specifying on-disk parameters on OST and MDT devices:

```
# tune2fs -o security.rpc.mdt=flavor1 -o security.rpc.cli=flavor2 \
device
```

On-disk parameters are overridden by mount options.

11.2.2.5 Mounting Clients

Root on client node mounts Lustre without any special tricks.

11.2.2.6 Rules, Syntax and Examples

The general rules and syntax for using Kerberos are:

```
<target>.srpc.flavor.<network>[.<direction>]=flavor
```

- <target>: This could be file system name or specific MDT/OST device name. For example, lustre, lustre-MDT0000, lustre-OST0001.
- <network>: LNET network name of the RPC initiator. For example, tcp0, elan1, o2ib0.
- <direction>: This could be one of cli2mdt, cli2ost, mdt2mdt, or mdt2ost. In most cases, you do not need to specify the <direction> part.

Examples:

- Apply ***krb5i*** on ALL connections:

```
mgs> lctl conf_param lustre.srpc.flavor.default=krb5i
```

- For nodes in network tcp0, use ***krb5p***. All other nodes use ***null***.

```
mgs> lctl conf_param lustre.srpc.flavor.tcp0=krb5p
mgs> lctl conf_param lustre.srpc.flavor.default=null
```

- For nodes in network tcp0, use ***krb5p***; for nodes in elan1, use ***plain***; Among other nodes, clients use ***krb5i*** to MDT/OST, MDT use ***null*** to other MDTs, MDT use ***plain*** to OSTs.

```
mgs> lctl conf_param lustre.srpc.flavor.tcp0=krb5p
mgs> lctl conf_param lustre.srpc.flavor.elan1=plain
mgs> lctl conf_param lustre.srpc.flavor.default.cli2mdt=krb5i
mgs> lctl conf_param lustre.srpc.flavor.default.cli2ost=krb5i
mgs> lctl conf_param lustre.srpc.flavor.default.mdt2mdt=null
mgs> lctl conf_param lustre.srpc.flavor.default.mdt2ost=plain
```

11.2.2.7 Authenticating Normal Users

On client nodes, non-root users must use **kinit** to access Lustre (just like other Kerberized applications). **kinit** is used to obtain and cache Kerberos ticket-granting tickets. Two requirements to authenticating users:

- Before **kinit** is run, the user must be registered as a principal with the Kerberos server (the Key Distribution Center or KDC). In KDC, the username is noted as username@REALM.
- The client and MDT nodes should have the same user database.

To destroy the established security contexts before logging out, run `lfs flushctx`:

```
# lfs flushctx [-k]
```

Here `-k` also means destroy the on-disk Kerberos credential cache. It is equivalent to `kdestroy`. Otherwise, it only destroys established contexts in the Lustre kernel.

CHAPTER 12

Bonding

This chapter describes how to set up bonding with Lustre, and includes the following sections:

12.1 Network Bonding

Bonding, also known as link aggregation, trunking and port trunking, is a method of aggregating multiple physical network links into a single logical link for increased bandwidth.

Several different types of bonding are supported in Linux. All these types are referred to as “modes,” and use the bonding kernel module.

Modes 0 to 3 provide support for load balancing and fault tolerance by using multiple interfaces. Mode 4 aggregates a group of interfaces into a single virtual interface where all members of the group share the same speed and duplex settings. This mode is described under IEEE spec 802.3ad, and it is referred to as either “mode 4” or “802.3ad.”

(802.3ad refers to mode 4 only. The detail is contained in Clause 43 of the IEEE 8 - the larger 802.3 specification. For more information, consult IEEE.)

12.2 Requirements

The most basic requirement for successful bonding is that both endpoints of the connection must support bonding. In a normal case, the non-server endpoint is a switch. (Two systems connected via crossover cables can also use bonding.) Any switch used must explicitly support 802.3ad Dynamic Link Aggregation.

The kernel must also support bonding. All supported Lustre kernels have bonding functionality. The network driver for the interfaces to be bonded must have the ethtool support. To determine slave speed and duplex settings, ethtool support is necessary. All recent network drivers implement it.

To verify that your interface supports ethtool, run:

```
# which ethtool
/sbin/ethtool

# ethtool eth0
Settings for eth0:
              Supported ports: [ TP MII ]
              Supported link modes:   10baseT/Half 10baseT/Full/
                                      100baseT/Half 100baseT/Full
              Supports auto-negotiation: Yes
              Advertised link modes:  10baseT/Half 10baseT/Full
                                      100baseT/Half 100baseT/Full
              Advertised auto-negotiation: Yes
              Speed: 100Mb/s
              Duplex: Full
              Port: MII
              PHYAD: 1
              Transceiver: internal
              Auto-negotiation: on
              Supports Wake-on: pumbg
              Wake-on: d
              Current message level: 0x00000001 (1)
              Link detected: yes
```

```
# ethtool eth1

Settings for eth1:
          Supported ports: [ TP MII ]
          Supported link modes:   10baseT/Half 10baseT/Full
                                  100baseT/Half 100baseT/Full
          Supports auto-negotiation: Yes
          Advertised link modes:  10baseT/Half 10baseT/Full
          100baseT/Half 100baseT/Full
          Advertised auto-negotiation: Yes
          Speed: 100Mb/s
          Duplex: Full
          Port: MII
          PHYAD: 32
          Transceiver: internal
          Auto-negotiation: on
          Supports Wake-on: pumbg
          Wake-on: d
          Current message level: 0x00000007 (7)
          Link detected: yes
          To quickly check whether your kernel supports bonding, run:
          # grep ifenslave /sbin/ifup
          # which ifenslave
          /sbin/ifenslave
```

Note – Bonding and ethtool have been available since 2000. All Lustre-supported kernels include this functionality.

12.3 Using Lustre with Multiple NICs versus Bonding NICs

Lustre can use multiple NICs without bonding. There is a difference in performance when Lustre uses multiple NICs versus when it uses bonding NICs.

Whether an aggregated link actually yields a performance improvement proportional to the number of links provided, depends on network traffic patterns and the algorithm used by the devices to distribute frames among aggregated links. Performance with bonding depends on:

- Out-of-order packet delivery

  This can trigger TCP congestion control. To avoid this, some bonding drivers restrict a single TCP conversation to a single adapter within the bonded group.

- Load balancing between devices in the bonded group.

  Consider a scenario with a two CPU node with two NICs. If the NICs are bonded, Lustre establishes a single bundle of sockets to each peer. Since ksocklnd bind sockets to CPUs, only one CPU moves data in and out of the socket for a uni-directional data flow to each peer. If the NICs are not bonded, Lustre establishes two bundles of sockets to the peer. Since ksocklnd spreads traffic between sockets, and sockets between CPUs, both CPUs move data.

12.4 Bonding Module Parameters

Bonding module parameters control various aspects of bonding.

Outgoing traffic is mapped across the slave interfaces according to the transmit hash policy. For Lustre, we recommend that you set the xmit_hash_policy option to the layer3+4 option for bonding. This policy uses upper layer protocol information if available to generate the hash. This allows traffic to a particular network peer to span multiple slaves, although a single connection does not span multiple slaves.

```
$ xmit_hash_policy=layer3+4
```

The `miimon` option enables users to monitor the link status. (The parameter is a time interval in milliseconds.) It makes an interface failure transparent to avoid serious network degradation during link failures. A reasonable default setting is 100 milliseconds; run:

```
$ miimon=100
```

For a busy network, increase the timeout.

12.5 Setting Up Bonding

To set up bonding:

1. **Create a virtual 'bond' interface by creating a configuration file in:**

   ```
   /etc/sysconfig/network-scripts/ # vi /etc/sysconfig/ \
   network-scripts/ifcfg-bond0
   ```

2. **Append the following lines to the file.**

   ```
   DEVICE=bond0
   IPADDR=192.168.10.79 # Use the free IP Address of your network
   NETWORK=192.168.10.0
   NETMASK=255.255.255.0
   USERCTL=no
   BOOTPROTO=none
   ONBOOT=yes
   ```

3. **Attach one or more slave interfaces to the bond interface. Modify the eth0 and eth1 configuration files (using a VI text editor).**

   a. **Use the VI text editor to open the eth0 configuration file.**

   ```
   # vi /etc/sysconfig/network-scripts/ifcfg-eth0
   ```

   b. **Modify/append the eth0 file as follows:**

   ```
   DEVICE=eth0
   USERCTL=no
   ONBOOT=yes
   MASTER=bond0
   SLAVE=yes
   BOOTPROTO=none
   ```

   c. **Use the VI text editor to open the eth1 configuration file.**

   ```
   # vi /etc/sysconfig/network-scripts/ifcfg-eth1
   ```

   d. **Modify/append the eth1 file as follows:**

   ```
   DEVICE=eth1
   USERCTL=no
   ONBOOT=yes
   MASTER=bond0
   SLAVE=yes
   BOOTPROTO=none
   ```

4. **Set up the bond interface and its options in /etc/modprobe.conf. Start the slave interfaces by your normal network method.**

   ```
   # vi /etc/modprobe.conf
   ```

   a. **Append the following lines to the file.**

   ```
   alias bond0 bonding
   options bond0 mode=balance-alb miimon=100
   ```

   b. **Load the bonding module.**

   ```
   # modprobe bonding
   # ifconfig bond0 up
   # ifenslave bond0 eth0 eth1
   ```

5. **Start/restart the slave interfaces (using your normal network method).**

Note – You must modprobe the bonding module for each bonded interface. If you wish to create bond0 and bond1, two entries in `modprobe.conf` are required.

The examples below are from RedHat systems. For setup use: `/etc/sysconfig/networking-scripts/ifcfg-*` The OSDL website referenced below includes detailed instructions for other configuration methods, instructions to use DHCP with bonding, and other setup details. We strongly recommend you use this website.

http://linux-net.osdl.org/index.php/Bonding

6. **Check /proc/net/bonding to determine status on bonding. There should be a file there for each bond interface.**

```
# cat /proc/net/bonding/bond0
Ethernet Channel Bonding Driver: v3.0.3 (March 23, 2006)

Bonding Mode: load balancing (round-robin)
MII Status: up
MII Polling Interval (ms): 0
Up Delay (ms): 0
Down Delay (ms): 0

Slave Interface: eth0
MII Status: up
Link Failure Count: 0
Permanent HW addr: 4c:00:10:ac:61:e0

Slave Interface: eth1
MII Status: up
Link Failure Count: 0
Permanent HW addr: 00:14:2a:7c:40:1d
```

7. **Use ethtool or ifconfig to check the interface state. ifconfig lists the first bonded interface as "bond0."**

```
ifconfig
bond0     Link encap:Ethernet  HWaddr 4C:00:10:AC:61:E0
          inet addr:192.168.10.79  Bcast:192.168.10.255 \
          Mask:255.255.255.0
          inet6 addr: fe80::4e00:10ff:feac:61e0/64 Scope:Link
          UP BROADCAST RUNNING MASTER MULTICAST  MTU:1500 Metric:1
          RX packets:3091 errors:0 dropped:0 overruns:0 frame:0
          TX packets:880 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:314203 (306.8 KiB)  TX bytes:129834 (126.7 KiB)

eth0      Link encap:Ethernet  HWaddr 4C:00:10:AC:61:E0
          inet6 addr: fe80::4e00:10ff:feac:61e0/64 Scope:Link
          UP BROADCAST RUNNING SLAVE MULTICAST  MTU:1500 Metric:1
          RX packets:1581 errors:0 dropped:0 overruns:0 frame:0
          TX packets:448 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:162084 (158.2 KiB)  TX bytes:67245 (65.6 KiB)
          Interrupt:193 Base address:0x8c00

eth1      Link encap:Ethernet  HWaddr 4C:00:10:AC:61:E0
          inet6 addr: fe80::4e00:10ff:feac:61e0/64 Scope:Link
          UP BROADCAST RUNNING SLAVE MULTICAST  MTU:1500 Metric:1
          RX packets:1513 errors:0 dropped:0 overruns:0 frame:0
          TX packets:444 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:152299 (148.7 KiB)  TX bytes:64517 (63.0 KiB)
          Interrupt:185 Base address:0x6000
```

12.5.1 Examples

This is an example of `modprobe.conf` for bonding Ethernet interfaces eth1 and eth2 to bond0:

```
# cat /etc/modprobe.conf
alias eth0 8139too
alias scsi_hostadapter sata_via
alias scsi_hostadapter1 usb-storage
alias snd-card-0 snd-via82xx
options snd-card-0 index=0
options snd-via82xx index=0
alias bond0 bonding
options bond0 mode=balance-alb miimon=100
options lnet networks=tcp
alias eth1 via-rhine

# cat /etc/sysconfig/network-scripts/ifcfg-bond0
DEVICE=bond0
BOOTPROTO=none
NETMASK=255.255.255.0
IPADDR=192.168.10.79 # (Assign here the IP of the bonded interface.)
ONBOOT=yes
USERCTL=no

ifcfg-ethx
# cat /etc/sysconfig/network-scripts/ifcfg-eth0
TYPE=Ethernet
DEVICE=eth0
HWADDR=4c:00:10:ac:61:e0
BOOTPROTO=none
ONBOOT=yes
USERCTL=no
IPV6INIT=no
PEERDNS=yes
MASTER=bond0
SLAVE=yes
```

In the following example, the bond0 interface is the master (MASTER) while eth0 and eth1 are slaves (SLAVE).

Note – All slaves of bond0 have the same MAC address (Hwaddr) – bond0. All modes, except TLB and ALB, have this MAC address. TLB and ALB require a unique MAC address for each slave.

```
$ /sbin/ifconfig

bond0Link encap:EthernetHwaddr 00:C0:F0:1F:37:B4
inet addr:XXX.XXX.XXX.YYY Bcast:XXX.XXX.XXX.255 Mask:255.255.252.0
UP BROADCAST RUNNING MASTER MULTICAST MTU:1500  Metric:1
RX packets:7224794 errors:0 dropped:0 overruns:0 frame:0
TX packets:3286647 errors:1 dropped:0 overruns:1 carrier:0
collisions:0 txqueuelen:0

eth0Link encap:EthernetHwaddr 00:C0:F0:1F:37:B4
inet addr:XXX.XXX.XXX.YYY Bcast:XXX.XXX.XXX.255 Mask:255.255.252.0
UP BROADCAST RUNNING SLAVE MULTICAST MTU:1500  Metric:1
RX packets:3573025 errors:0 dropped:0 overruns:0 frame:0
TX packets:1643167 errors:1 dropped:0 overruns:1 carrier:0
collisions:0 txqueuelen:100
Interrupt:10 Base address:0x1080

eth1Link encap:EthernetHwaddr 00:C0:F0:1F:37:B4
inet addr:XXX.XXX.XXX.YYY Bcast:XXX.XXX.XXX.255 Mask:255.255.252.0
UP BROADCAST RUNNING SLAVE MULTICAST MTU:1500  Metric:1
RX packets:3651769 errors:0 dropped:0 overruns:0 frame:0
TX packets:1643480 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:100
Interrupt:9 Base address:0x1400
```

12.6 Configuring Lustre with Bonding

Lustre uses the IP address of the bonded interfaces and requires no special configuration. It treats the bonded interface as a regular TCP/IP interface. If needed, specify “bond0” using the Lustre networks parameter in `/etc/modprobe`

```
options lnet networks=tcp(bond0)
```

12.6.1 Bonding References

We recommend the following bonding references:

In the Linux kernel source tree, see `documentation/networking/bonding.txt`

http://linux-ip.net/html/ether-bonding.html

http://www.sourceforge.net/projects/bonding

This is the bonding SourceForge website:

http://linux-net.osdl.org/index.php/Bonding

This is the most extensive reference and we highly recommend it. This website includes explanations of more complicated setups, including the use of DHCP with bonding.

CHAPTER 13

Upgrading and Downgrading Lustre

The chapter describes how to upgrade and downgrade between different Lustre versions and includes the following sections:

13.1 Supported Upgrades

For Lustre 1.8.x, the following upgrades are supported:

- Lustre 1.6.x (latest version) to Lustre 1.8.x (latest version)
- Lustre 1.8.x (any minor version) to Lustre 1.8.x (latest version)

13.2 Lustre Interoperability

Lustre interoperability enables 1.8.x servers ("new" servers) to work with 1.6.x clients ("old" clients), 1.6.x servers ("old" servers) to work with 1.8.x clients ("new" clients), and "mixed" environments with 1.6.x and 1.8.x servers. For example, half of each OSS failover pair could be upgraded to enable a quick reversion to 1.6 by powering down the 1.8 servers.

This table describes interoperability between Lustre clients, OSTs and MDTs with different versions of Lustre installed.

Lustre Component	Interoperability with Other Lustre Components
Clients	• Old, live clients can communicate with old/new/mixed servers • Old clients can start up using old/new/mixed servers • New clients can start up using old/new/mixed servers **Note -** Old clients cannot mount a file system that was created by a new MDT.
OSTs	• Old OSTs can communicate with new clients/MDT • New OSTs can only be started after the MGS has been started (typically this means "after the MDT has been upgraded")
MDTs	• Old MDT can communicate with new clients • New, co-located MGS/MDT can be started at any point • New, non co-located MDT can be started after the MGS starts

13.3 Upgrading Lustre 1.6.x to 1.8.x

Two upgrade paths are supported to meet the upgrade requirements of different Lustre environments.

- **Complete file system** - All servers and clients are shut down and upgraded at the same time. See Performing a Complete File System Upgrade.
- **Rolling upgrade** - Individual servers (or their failover partners) and clients are upgraded one at a time, so the file system never goes down. See Performing a Rolling Upgrade.

Note – If you upgrade some Lustre components to 1.8.x but not others (such as running 1.8 clients in a file system with 1.6 OSTs), and run a mixed environment, you may see one or more warnings similar to this:

```
LustreError: 3877:0:(socklnd_cb.c:2228:ksocknal_recv_hello())
Unknown protocol version (2.x expected) from 192.168.2.43
```

This warning is given when the 1.6 and 1.8 components use different protocols. It can be safely ignored because the Lustre components negotiate a common protocol. In this example, the 1.8 clients fall back to use the 1.6 protocol with the 1.6 OSTs.

13.3.1 Performing a Complete File System Upgrade

This procedure describes a complete file system upgrade in which 1.8.x Lustre packages are installed on multiple 1.6.x servers and clients, requiring a file system shut down. If you want to upgrade one Lustre component at a time and avoid the shutdown, see Performing a Rolling Upgrade.

Tip – In a Lustre upgrade, the package install and file system unmount steps are reversible; you can do either step first. To minimize downtime, this procedure first performs the 1.8.x package installation, and then unmounts the file system.

---

1. **Make a complete, restorable file system backup before upgrading Lustre.**
2. **Install the 1.8.x packages on the Lustre servers and/or clients.**

   Some or all servers can be upgraded. Some or all clients can be upgraded.

   For help determining where to install a specific package, see TABLE 3-1 (Lustre packages, descriptions and installation guidance).

   a. **Install the kernel, modules and ldiskfs packages. For example:**

   ```
   $ rpm -ivh
   kernel-lustre-smp-<ver> \
   kernel-ib-<ver> \
   lustre-modules-<ver> \
   lustre-ldiskfs-<ver>
   ```

   b. **Upgrade the utilities/userspace packages. For example:**

   ```
   $ rpm -Uvh lustre-<ver>
   ```

   c. **If a new e2fsprogs package is available, upgrade it. For example:**

   ```
   $ rpm -Uvh e2fsprogs-<ver>
   ```

   There may or may not be a new e2fsprogs package with a Lustre upgrade. The e2fsprogs release schedule is independent of Lustre releases.

   d. **(Optional) If you want to add optional packages to your Lustre system, install them now.**

3. **Shut down the file system.**

   Shut down the components in this order: clients, then the MDT, then OSTs. Unmounting a block device causes Lustre to be shut down on that node.

   a. **Unmount the clients. On each client node, run:**

      ```
      umount <mount point>
      ```

   b. **Unmount the MDT. On the MDS node, run:**

      ```
      umount <mount point>
      ```

   c. **Unmount the OSTs (be sure to unmount all OSTs). On each OSS node, run:**

      ```
      umount <mount point>
      ```

4. **Unload the old Lustre modules by either:**

   - Rebooting the node

   - OR -

   - Removing the Lustre modules manually. Run `lustre_rmmod` several times and use `lsmod` to check the currently loaded modules.

5. **Start the upgraded file system.**

   Start the components in this order: OSTs, then the MDT, then clients.

   a. **Mount the OSTs (be sure to mount all OSTs). On each OSS node, run:**

      ```
      mount -t lustre <block device name> <mount point>
      ```

   b. **Mount the MDT. On the MDS node, run:**

      ```
      mount -t lustre <block device name> <mount point>
      ```

   c. **Mount the file system on the clients. On each client node, run:**

      ```
      mount -t lustre <MGS node>:/<fsname> <mount point>
      ```

If you have a problem upgrading Lustre, contact us via the Bugzilla bug tracker.

13.3.2 Performing a Rolling Upgrade

This procedure describes a rolling upgrade in which one Lustre component (server or client) is upgraded and restarted at a time while the file system is running. If you want to upgrade the complete Lustre file system or multiple components at a time, requiring a file system shutdown, see Performing a Complete File System Upgrade.

Note – The suggested upgrade order is the MGS first, then OSTs, then the MDT, and then clients. These are general guidelines, and specific upgrade requirements can be found in the release notes for a given Lustre version. If no particular restrictions are stated, then the suggested upgrade order may be rearranged; bear in mind that the suggested order is the most heavily tested by the Lustre team.

Note – If the Lustre component to be upgraded is an OSS in a failover pair, follow these special upgrade steps to minimize downtime:

1. Fail over the server to its peer server, so the file system remains available.

2. Install the Lustre 1.8.x packages on the idle server.

3. Unload the old Lustre modules on the idle server by either:

Rebooting the node

- OR -

Removing the Lustre modules manually by running the `lustre_rmmod` command several times and checking the currently loaded modules with the `lsmod` command.

4. Fail back services to the idle (now upgraded) server.

5. Repeat Steps 1 to 4 on the peer server.

This limits the outage (per OSS) to a single server for as long as it takes to fail over.

---

1. **Make a complete, restorable file system backup before upgrading Lustre.**

2. **Install the 1.8.x packages on the Lustre component (server or client).**

   For help determining where to install a specific package, see TABLE 3-1 (Lustre packages, descriptions and installation guidance).

   a. **Install the kernel, modules and ldiskfs packages. For example:**

   ```
   $ rpm -ivh
   kernel-lustre-smp-<ver> \
   kernel-ib-<ver> \
   lustre-modules-<ver> \
   lustre-ldiskfs-<ver>
   ```

   b. **Upgrade the utilities/userspace packages. For example:**

   ```
   $ rpm -Uvh lustre-<ver>
   ```

   c. **If a new e2fsprogs package is available, upgrade it. For example:**

   ```
   $ rpm -Uvh e2fsprogs-<ver>
   ```

   There may or may not be a new e2fsprogs package with a Lustre upgrade. The e2fsprogs release schedule is independent of Lustre releases.

   d. **(Optional) If you want to add optional packages to your Lustre system, install them now.**

3. **Unload the old Lustre modules by either:**

   - Rebooting the node

     - OR -

   - Removing the Lustre modules manually. Run `lustre_rmmod` several times and use `lsmod` to check the currently-loaded modules.

4. **If the upgraded component is a server, fail back services to it.**

If you have a problem upgrading Lustre, contact us via the Bugzilla bug tracker.

13.4 Upgrading Lustre 1.8.x to the Next Minor Version

To upgrade Lustre 1.8.x to the next minor version, for example, Lustre 1.8.0.1 > 1.8.x, follow these procedures:

- To upgrade the complete file system or multiple file system components at the same time, requiring a file system shutdown, see Performing a Complete File System Upgrade
- To upgrade one Lustre component (server or client) at a time, while the file system is running, see Performing a Rolling Upgrade

13.5 Downgrading from Lustre 1.8.x to 1.6.x

This section describes how to downgrade from Lustre 1.8.x to 1.6.x. Only file systems that were upgraded from 1.6.x can be downgraded to 1.6.x. A file system that was created or reformatted under Lustre 1.8.x cannot be downgraded.

Two paths are available to meet the downgrade requirements of different Lustre environments.

- **Complete file system** - File system is shut down and all servers and clients are downgraded at once. See Performing a Complete File System Downgrade.
- **Individual servers / clients** - Individual servers and clients are downgraded one at a time and restarted (a "rolling downgrade"), so the file system never goes down. See Performing a Rolling Downgrade.

13.5.1 Performing a Complete File System Downgrade

This procedure describes a complete file system downgrade in which 1.6.x Lustre packages are installed on multiple 1.8.x servers and clients, requiring a file system shut down. If you want to upgrade one Lustre component at a time and avoid the shutdown, see Performing a Rolling Downgrade.

Tip – In a Lustre downgrade, the package install and file system unmount steps are reversible; you can do either step first. To minimize downtime, this procedure first performs the 1.6.x package installation, and then unmounts the file system.

---

1. **Make a complete, restorable file system backup before downgrading Lustre.**
2. **Verify that 1.6.x packages are installed on the Lustre servers and/or clients.**
   a. **Check that the kernel, modules and ldiskfs packages are installed.**

      The 1.6.x kernel, modules and ldiskfs packages should be on all nodes because of the earlier upgrade to 1.8.x, unless they were removed after the upgrade.

      If it is necessary to install kernel, modules or ldiskfs packages, use the `rpm -ivh` command. For example:

      ```
      $ rpm -ivh
      kernel-lustre-smp-<ver> \
      kernel-ib-<ver> \
      lustre-modules-<ver> \
      lustre-ldiskfs-<ver>
      ```

      For help determining where to install a specific package, see TABLE 3-1 (Lustre packages, descriptions and installation guidance).

   b. **Install the utilities/userspace packages, using the `--oldpackage` option. For example:**

      ```
      rpm -Uvh --oldpackage lustre-<ver>
      ```

Note – You do not need to downgrade or take any action with e2fsprogs.

---

3. **Shut down the file system.**

   Shut down the components in this order: clients, then the MDT, then OSTs. Unmounting a block device causes Lustre to be shut down on that node.

   a. **Unmount the clients. On each client node, run:**

      ```
      umount <mount point>
      ```

   b. **Unmount the MDT. On the MDS node, run:**

      ```
      umount <mount point>
      ```

   c. **Unmount the OSTs (be sure to unmount all OSTs). On each OSS node, run:**

      ```
      umount <mount point>
      ```

4. **Unload the old Lustre modules by either:**

   - Rebooting the node

   - OR -

   - Removing the Lustre modules manually. Run `lustre_rmmod` several times and use `lsmod` to check the currently loaded modules.

5. **Start the downgraded file system.**

   Start the components in this order: OSTs, then the MDT, then clients.

   a. **Mount the OSTs (be sure to mount all OSTs). On each OSS node, run:**

      ```
      mount -t lustre <block device name> <mount point>
      ```

   b. **Mount the MDT. On the MDS node, run:**

      ```
      mount -t lustre <block device name> <mount point>
      ```

   c. **Mount the file system on the clients. On each client node, run:**

      ```
      mount -t lustre <MGS node>:/<fsname> <mount point>
      ```

If you have a problem downgrading Lustre, contact us via the Bugzilla bug tracker.

13.5.2 Performing a Rolling Downgrade

This procedure describes a rolling downgrade in which one Lustre component (server or client) is downgraded and restarted at a time while the file system is running. If you want to downgrade the complete Lustre file system or multiple components at a time, requiring a file system shutdown, see Performing a Complete File System Downgrade.

Note – If the Lustre component to be downgraded is an OSS in a failover pair, follow these special downgrade steps to minimize downtime:

1. Fail over the server to its peer server, so the file system remains available.

2. Install the Lustre 1.8.x packages on the idle server.

3. Unload the old Lustre modules on the idle server by either:

Rebooting the node

- OR -

Removing the Lustre modules manually by running the `lustre_rmmod` command several times and checking the currently loaded modules with the `lsmod` command.

4. Fail back services to the idle (now upgraded) server.

5. Repeat Steps 1 to 4 on the peer server.

This limits the outage (per OSS) to a single server for as long as it takes to fail over.

---

1. **Make a complete, restorable file system backup before downgrading Lustre.**

2. **Install the 1.6.x packages on the Lustre component (server or client).**

   For help determining where to install a specific package, see TABLE 3-1 (Lustre packages, descriptions and installation guidance).

   a. **Install the kernel, modules and ldiskfs packages. For example:**

   ```
   $ rpm -ivh
   kernel-lustre-smp-<ver> \
   kernel-ib-<ver> \
   lustre-modules-<ver> \
   lustre-ldiskfs-<ver>
   ```

   b. **Downgrade the utilities/userspace packages, using the `--oldpackage` option. For example:**

   ```
   $ rpm -Uvh --oldpackage lustre-<ver>
   ```

Note – You do not need to downgrade or take any action with e2fsprogs.

---

3. **Unload the old Lustre modules by either:**

   - Rebooting the node

     - OR -

   - Removing the Lustre modules manually. Run `lustre_rmmod` several times and use `lsmod` to check the currently-loaded modules.

4. **If the upgraded component is a server, fail back services to it.**

If you have a problem upgrading Lustre, contact us via the Bugzilla bug tracker.

CHAPTER 14

Lustre SNMP Module

The Lustre SNMP module reports information about Lustre components and system status, and generates traps if an LBUG occurs. The Lustre SNMP module works with the net-snmp. The module consists of a plug-in (lustresnmp.so), which is loaded by the snmpd daemon, and a MIB file (`Lustre-MIB.txt`).

This chapter describes how to install and use the Lustre SNMP module, and includes the following sections:

14.1 Installing the Lustre SNMP Module

To install the Lustre SNMP module:

1. **Locate the SNMP plug-in (lustresnmp.so) in the base Lustre RPM and install it.**

   ```
   /usr/lib/lustre/snmp/lustresnmp.so
   ```

2. **Locate the MIB (Lustre-MIB.txt) in /usr/share/lustre/snmp/mibs/Lustre-MIB.txt and append the following line to snmpd.con.**

   ```
   dlmod lustresnmp /usr/lib/lustre/snmp/lustresnmp.so
   ```

3. **You may need to copy Lustre-MIB.txt to a different location to use few tools. For this, use either of these commands.**

   ```
   ~/.snmp/mibs
   /usr/local/share/snmp/mibs
   ```

14.2 Building the Lustre SNMP Module

To build the Lustre SNMP module, you need the `net-snmp-devel` package. The default `net-snmp` install includes a `snmpd.conf` file.

1. **Complete the net-snmp setup by checking and editing the snmpd.conf file, located in /etc/snmp**

   ```
   /etc/snmp/snmpd.conf
   ```

2. **Build the Lustre SNMP module from the Lustre src.rpm**
   - Install the Lustre source
   - Run ./configure
   - Add the `--enable-snmp` option

14.3 Using the Lustre SNMP Module

Once the Lustre SNMP module in installed and built, use it for purposes:

- For all Lustre components, the SNMP module reports a number and total and free capacity (usually in bytes).
- Depending on the component type, SNMP also reports total or free numbers for objects like OSD and OSC or other files (LOV, MDC, and so on).
- The Lustre SNMP module provides one read/write variable, sysStatus, which starts and stops Lustre.
- The sysHealthCheck object reports status either as healthy' or 'not healthy' and provides information for the failure.
- The Lustre SNMP module generates traps on the detection of LBUG (lustrePortalsCatastropeTrap), and detection of various OBD-specific healthchecks (lustreOBDUnhealthyTrap).

CHAPTER 15

Backup and Restore

Lustre provides backups at the file system-level, device-level and file-level. This chapter describes how to backup and restore on Lustre, and includes the following sections:

15.1 Backing up a File System

Backing up a complete file system gives you full control over the files to back up, and allows restoration of individual files as needed. File system-level backups are also the easiest to integrate into existing backup solutions.

File system backups are performed from a Lustre client (or many clients working parallel in different directories) rather than on individual server nodes; this is no different than backing up any other file system.

However, due to the large size of most Lustre file systems, it is not always possible to get a complete backup. We recommend that you back up subsets of a file system. This includes subdirectories of the entire file system, filesets for a single user, files incremented by date, and so on.

15.2 Backing up a Device (MDS or OST)

In some cases, it is useful to do a full, device-level backup of an individual device (MDS or OST), before replacing hardware, performing maintenance, etc. Doing full device-level backups ensures that all of the data is preserved in the original state and is the easiest method of doing a backup.

Note – A device-level backup of the MDS is especially important because, if it fails permanently, the entire file system would need to be restored.

If hardware replacement is the reason for the backup or if a spare storage device is available, it is possible to do a raw copy of the MDS or OST from one block device to the other, as long as the new device is at least as large as the original device. To do this, run:

```
dd if=/dev/{original} of=/dev/{new} bs=1M
```

If hardware errors cause read problems on the original device, use the command below to allow as much data as possible to be read from the original device while skipping sections of the disk with errors:

```
dd if=/dev/{original} of=/dev/{new} bs=4k conv=sync,noerror count=
{original size in 4kB blocks}
```

Even in the face of hardware errors, the ext3 file system is very robust and it may be possible to recover the file system data after running `e2fsck -f` on the new device.

15.2.1 Backing Up the MDS

This procedure provides another way to back up the MDS.

1. **Make a mount point for the file system. Run:**

   ```
   mkdir -p /mnt/mds
   ```

2. **Mount the file system. Run:**

   ```
   mount -t ldiskfs {mdsdev} /mnt/mds
   ```

3. **Change to the mount point being backed up. Run:**

   ```
   cd /mnt/mds
   ```

4. **Back up the EAs. Run:**

```
getfattr -R -d -m '.*' -P . > ea.bak
```

Note – In most distributions, the `getfattr` command is part of the "attr" package. If the `getfattr` command returns errors like `Operation not supported`, then the kernel does not correctly support EAs. Stop and use a different backup method or contact us for assistance.

---

5. **Verify that the ea.bak file has properly backed up the EA data on the MDS. Without this EA data, the backup is not useful. Look at this file with "more" or a text editor. For each file, it should have an item similar to this:**

```
# file: ROOT/mds_md5sum3.txt
trusted.lov=
0s0AvRCwEAAABXoKUCAAAAAAAAAAAAAAAAAAAQAAEAAADD5QoAAAAAAAAAAAAAAAAA
AAAAAAEAAAA=
```

6. **Back up all file system data. Run:**

```
tar czvf {backup file}.tgz --sparse .
```

Note – In Lustre 1.6.7 and later, the `--sparse` option reduces the size of the backup file. Be sure to use it in the tar command.

---

7. **Change directory out of the mounted file system. Run:**

```
cd -
```

8. **Unmount the file system. Run:**

```
umount /mnt/mds
```

Note – When restoring an MDT backup on a different node as part of an MDT migration, you also have to change server NIDs and use the `--writeconf` command to re-generate the configuration logs. See Changing a Server NID and osc.myth-OST0004-osc-ffff88006dd20000.filesfree=129651.

15.2.2 Backing Up an OST

Follow the same procedure as Backing Up the MDS (except skip Step 5) and, for each OST device file system, replace `mds` with `ost` in the commands.

15.3 Backing up Files

In other cases, it is desirable to back up only the file data on an MDS or OST instead of the entire device, e.g., if the device is very large but has little data in it, if the configuration of the parameters of the ext3 filesystem need to be changed, to use less space for the backup, etc.

In this situation, it is possible to mount the ext3 filesystem directly from the storage device, and do a file-level backup. Lustre **MUST STOP** be stopped on this node.

15.3.1 Backing up Extended Attributes

In Lustre, each OST object has an extended attribute (EA) that contains the MDT inode number and stripe index for the object. The EA's striping information includes the location of file data on the OSTs and OST pool membership. The EA data must be backed up or the file backup will not be useful. Current backup tools do not properly save the EA data, so the following extra steps are required.

1. **Make a mountpoint for the file system.**

   ```
   mkdir /mnt/mds
   ```

2. **Mount the filesystem.**

   ```
   mount -t ldiskfs {olddev} /mnt/mds
   ```

3. **Change to the mountpoint being backed up.**

   ```
   cd /mnt/mds
   ```

4. **Back up the extended attributes.**

   ```
   getfattr -R -d -m '.*' -P . > ea.bak
   ```

In most distributions, the `getfattr` command is part of the "attr" package. If the `getfattr` command returns errors like "`Operation not supported`", then your kernel does not support EAs correctly. Stop and use a different backup method or submit a Bugzilla ticket.

5. **Verify that the ea.bak file has properly backed up the EA data on the MDS. You can look at this file with "more" or a text editor. For each file, it should have an item similar to this**

   ```
   # file: ROOT/mds_md5sum3.txt
   trusted.lov=0s0AvRCwEAAABXoKUCAAAAAAAAQAAEAAADD5QoAAAAAAAAAAEAAAA=
   ```

6. **Back up all file system data.**

   ```
   tar czvf {backup file}.tgz --sparse .
   ```

7. **Change out of the mounted file system.**

   ```
   cd -
   ```

8. **Unmount the file system.**

   ```
   umount /mnt/mds
   ```

9. **Print the file system label and write it down.**

   ```
   e2label {olddev}
   ```

The same process should be followed on each MDS or OST file system.

15.4 Restoring from a File-level Backup

To restore data from a file-level backup, you need to format the device, restore the file data and then restore the EA data.

1. **Format the new device. Run:**

   ```
   mkfs.lustre {--mdt|--ost} {other options} {newdev}
   ```

2. **Mount the file system. Run:**

   ```
   mount -t ldiskfs {newdev} /mnt/mds
   ```

3. **Change to the new file system mount point. Run:**

   ```
   cd /mnt/mds
   ```

4. **Restore the file system backup. Run:**

   ```
   tar xzvpf {backup file} --sparse
   ```

5. **Restore the file system extended attributes. Run:**

   ```
   setfattr --restore=ea.bak
   ```

6. **Verify that the extended attributes were restored. If this is not correct, then all data in the files will be lost, and would show up as all files in the filesystem having zero length.**

   ```
   getfattr -d -m ".*" ROOT/mds_md5sum3.txt
   trusted.lov=0s0AvRCwEAAABXoKUCAAAAAAAAQAAEAAADD5QoAAAAAAAAAEAAAA=
   ```

7. **Remove the (now invalid) recovery logs. Run:**

```
rm OBJECTS/* CATALOGS
```

8. **Change out of the MDS file system.**

```
cd -
```

9. **Unmount the MDS file system.**

```
umount /mnt/mds
```

If the file system was used between the time the backup was made and when it was restored, then the `lfsck` tool (part of Lustre e2fsprogs) can be run to ensure the file system is coherent. If all of the device file systems were backed up at the same time after the entire Lustre file system was stopped, this is not necessary. The file system should be immediately usable even if `lfsck` is not run, though there will be I/O errors reading from files that are present on the MDS but not the OSTs, and files that were created after the MDS backup will not be accessible/visible.

15.5 Using LVM Snapshots with Lustre

If you want to perform disk-based backups (because, for example, access to the backup system needs to be as fast as to the primary Lustre file system), you can use the Linux LVM snapshot tool to maintain multiple, incremental file system backups.

Because LVM snapshots cost CPU cycles as new files are written, taking snapshots of the main Lustre file system will probably result in unacceptable performance losses. You should create a new, backup Lustre file system and periodically (e.g., nightly) back up new/changed files to it. Periodic snapshots can be taken of this backup file system to create a series of "full" backups.

Note – Creating an LVM snapshot is not as reliable as making a separate backup, because the LVM snapshot shares the same disks as the primary MDT device, and depends on the primary MDT device for much of its data. If the primary MDT device becomes corrupted, this may result in the snapshot being corrupted.

15.5.1 Creating an LVM-based Backup File System

Use this procedure to create a backup Lustre file system for use with the LVM snapshot mechanism.

1. **Create LVM volumes for the MDT and OSTs.**

   Create LVM devices for your MDT and OST targets. Make sure not to use the entire disk for the targets; save some room for the snapshots. The snapshots start out as 0 size, but grow as you make changes to the current file system. If you expect to change 20% of the file system between backups, the most recent snapshot will be 20% of the target size, the next older one will be 40%, etc. Here is an example:

```
cfs21:~# pvcreate /dev/sda1
    Physical volume "/dev/sda1" successfully created
cfs21:~# vgcreate volgroup /dev/sda1
    Volume group "volgroup" successfully created
cfs21:~# lvcreate -L200M -nMDT volgroup
    Logical volume "MDT" created
cfs21:~# lvcreate -L200M -nOST0 volgroup
    Logical volume "OST0" created
cfs21:~# lvscan
    ACTIVE           '/dev/volgroup/MDT' [200.00 MB] inherit
    ACTIVE           '/dev/volgroup/OST0' [200.00 MB] inherit
```

2. **Format the LVM volumes as Lustre targets.**

   In this example, the backup file system is called “main” and designates the current, most up-to-date backup.

```
cfs21:~# mkfs.lustre --mdt --fsname=main /dev/volgroup/MDT
 No management node specified, adding MGS to this MDT.
    Permanent disk data:
 Target:     main-MDTffff
 Index:      unassigned
 Lustre FS:  main
 Mount type: ldiskfs
 Flags:      0x75
               (MDT MGS needs_index first_time update )
 Persistent mount opts: errors=remount-ro,iopen_nopriv,user_xattr
 Parameters:
checking for existing Lustre data
 device size = 200MB
 formatting backing filesystem ldiskfs on /dev/volgroup/MDT
         target name  main-MDTffff
         4k blocks     0
         options        -i 4096 -I 512 -q -O dir_index -F
 mkfs_cmd = mkfs.ext2 -j -b 4096 -L main-MDTffff  -i 4096 -I 512 -q
-O dir_index -F /dev/volgroup/MDT
 Writing CONFIGS/mountdata
cfs21:~# mkfs.lustre --ost --mgsnode=cfs21 --fsname=main
/dev/volgroup/OST0
    Permanent disk data:
 Target:     main-OSTffff
Index:      unassigned
 Lustre FS:  main
 Mount type: ldiskfs
 Flags:      0x72
               (OST needs_index first_time update )
 Persistent mount opts: errors=remount-ro,extents,mballoc
 Parameters: mgsnode=192.168.0.21@tcp
checking for existing Lustre data
 device size = 200MB
 formatting backing filesystem ldiskfs on /dev/volgroup/OST0
         target name  main-OSTffff
         4k blocks     0
         options        -I 256 -q -O dir_index -F
 mkfs_cmd = mkfs.ext2 -j -b 4096 -L main-OSTffff  -I 256 -q -O
dir_index -F /dev/ volgroup/OST0
 Writing CONFIGS/mountdata
cfs21:~# mount -t lustre /dev/volgroup/MDT /mnt/mdt
```

```
cfs21:~# mount -t lustre /dev/volgroup/OST0 /mnt/ost
cfs21:~# mount -t lustre cfs21:/main /mnt/main
```

15.5.2 Backing up New/Changed Files to the Backup File System

At periodic intervals e.g., nightly, back up new and changed files to the LVM-based backup file system.

```
cfs21:~# cp /etc/passwd /mnt/main

cfs21:~# cp /etc/fstab /mnt/main

cfs21:~# ls /mnt/main
fstab  passwd
```

15.5.3 Creating Snapshot Volumes

Whenever you want to make a "checkpoint" of the main Lustre file system, create LVM snapshots of all target MDT and OSTs in the LVM-based backup file system. You must decide the maximum size of a snapshot ahead of time, although you can dynamically change this later. The size of a daily snapshot is dependent on the amount of data changed daily in the main Lustre file system. It is likely that a two-day old snapshot will be twice as big as a one-day old snapshot.

You can create as many snapshots as you have room for in the volume group. If necessary, you can dynamically add disks to the volume group.

The snapshots of the target MDT and OSTs should be taken at the same point in time. Make sure that the cronjob updating the backup file system is not running, since that is the only thing writing to the disks. Here is an example:

```
cfs21:~# modprobe dm-snapshot
cfs21:~# lvcreate -L50M -s -n MDTb1 /dev/volgroup/MDT
   Rounding up size to full physical extent 52.00 MB
   Logical volume "MDTb1" created
cfs21:~# lvcreate -L50M -s -n OSTb1 /dev/volgroup/OST0
   Rounding up size to full physical extent 52.00 MB
   Logical volume "OSTb1" created
```

After the snapshots are taken, you can continue to back up new/changed files to "main". The snapshots will not contain the new files.

```
cfs21:~# cp /etc/termcap /mnt/main
cfs21:~# ls /mnt/main
fstab  passwd  termcap
```

15.5.4 Restoring the File System From a Snapshot

Use this procedure to restore the file system from an LVM snapshot.

1. **Rename the LVM snapshot.**

   Rename the file system snapshot from "main" to "back" so you can mount it without unmounting "main". This is recommended, but not required. Use the `--reformat` flag to `tunefs.lustre` to force the name change. For example:

```
cfs21:~# tunefs.lustre --reformat --fsname=back --writeconf
/dev/volgroup/MDTb1
 checking for existing Lustre data
 found Lustre data
 Reading CONFIGS/mountdata
Read previous values:
 Target:     main-MDT0000
 Index:      0
 Lustre FS:  main
 Mount type: ldiskfs
 Flags:      0x5
              (MDT MGS )
 Persistent mount opts: errors=remount-ro,iopen_nopriv,user_xattr
 Parameters:
Permanent disk data:
 Target:     back-MDT0000
 Index:      0
 Lustre FS:  back
 Mount type: ldiskfs
 Flags:      0x105
              (MDT MGS writeconf )
 Persistent mount opts: errors=remount-ro,iopen_nopriv,user_xattr
 Parameters:
Writing CONFIGS/mountdata
cfs21:~# tunefs.lustre --reformat --fsname=back --writeconf
/dev/volgroup/OSTb1
 checking for existing Lustre data
 found Lustre data
```

```
 Reading CONFIGS/mountdata
Read previous values:
 Target:     main-OST0000
 Index:      0
 Lustre FS:  main
 Mount type: ldiskfs
 Flags:      0x2
              (OST )
 Persistent mount opts: errors=remount-ro,extents,mballoc
 Parameters: mgsnode=192.168.0.21@tcp
Permanent disk data:
 Target:     back-OST0000
 Index:      0
 Lustre FS:  back
 Mount type: ldiskfs
 Flags:      0x102
              (OST writeconf )
 Persistent mount opts: errors=remount-ro,extents,mballoc
 Parameters: mgsnode=192.168.0.21@tcp
Writing CONFIGS/mountdata
When renaming an FS, we must also erase the last_rcvd file from the
snapshots
cfs21:~# mount -t ldiskfs /dev/volgroup/MDTb1 /mnt/mdtback
 cfs21:~# rm /mnt/mdtback/last_rcvd
 cfs21:~# umount /mnt/mdtback
 cfs21:~# mount -t ldiskfs /dev/volgroup/OSTb1 /mnt/ostback
 cfs21:~# rm /mnt/ostback/last_rcvd
 cfs21:~# umount /mnt/ostback
```

2. **Mount the file system from the LVM snapshot.**

   For example:

```
cfs21:~#mount-tlustre/dev/volgroup/MDTb1/mnt/mdtback
 cfs21:~# mount -t lustre /dev/volgroup/OSTb1 /mnt/ostback
 cfs21:~# mount -t lustre cfs21:/back /mnt/back
```

3. **Note the old directory contents, as of the snapshot time.**

   For example:

```
cfs21:~/cfs/b1_5/lustre/utils# ls /mnt/back
 fstab  passwds
```

15.5.5 Deleting Old Snapshots

To reclaim disk space, you can erase old snapshots as your backup policy dictates. Run:

```
lvremove /dev/volgroup/MDTb1
```

15.5.6 Changing Snapshot Volume Size

You can also extend or shrink snapshot volumes if you find your daily deltas are smaller or larger than expected. Run:

```
lvextend -L10G /dev/volgroup/MDTb1
```

Note – Extending snapshots seems to be broken in older LVM. It is working in LVM v2.02.01.

CHAPTER 16

POSIX

This chapter describes how to install and run the POSIX compliance suite of file system tests and includes the following sections:

16.1 Introduction to POSIX

Portable Operating System Interface (POSIX) is a set of standard, operating system interfaces based on the Unix OS. POSIX defines file system behavior on single UNIX node. Although used mainly with UNIX systems, the POSIX standard can apply to any operating system.

POSIX specifies the user and software interfaces to the OS. Required program-level services include basic I/O (file, terminal, and network) services. POSIX also defines a standard threading library API which is supported by most modern operating systems.

POSIX in a cluster means that most of the operations are atomic. Clients cannot see the metadata. POSIX offers strict mandatory locking which gives guarantee of semantics. Users do not have control on these locks.

Note – Lustre is not completely POSIX-compliant, so test results may show some errors. If you have questions about test results, contact our QE and Test Team (lustre-koala-team@sun.com).

16.2 Installing POSIX

Several quick start versions of the POSIX compliance suite are available to download. Each version is gcc- and architecture-specific. You need to determine which version of gcc you are running locally ({{{gcc -v}}}) and then download the appropriate tarball.

If a package is not available for your particular combination of gcc+architecture, see Building and Running a POSIX Compliance Test Suite on Lustre.

The following quick start versions are provided:

- one-step-gcc2.96-i686.tgz
- one-step-gcc2.96-ia64.tgz
- one-step-gcc3.04-i686.tgz
- one-step-gcc3.2-i686.tgz

16.2.1 POSIX Installation Using a Quick Start Version

Use this procedure to install POSIX using a quick start version.

1. **Download the POSIX scripts into** `/usr/src/posix`**.**
   - Test script: `one-step-gcc<gcc version>-<arch>.tgz`
   - Quick start script: `one-step-setup.sh`

   Both scripts are available at:

   http://downloads.lustre.org/public/tools/benchmarks/posix/
2. **Launch the setup script. Run:**

   ```
   cd /usr/src/posix
   sh one-step-setup.sh
   ```
3. **Edit the configuration file** `/mnt/lustre/TESTROOT/tetexec.cfg` **with appropriate values for your system.**

4. **Save the TESTROOT for running Lustre tests. Run:**

```
cd /mnt/lustre
tar zcvf /usr/src/posix/TESTROOT.tgz TESTROOT
```

Note – The quick start installation procedure only works with the paths `/home/tet` and `/mnt/lustre`. If you want to change the paths, follow the steps in Building and Running a POSIX Compliance Test Suite on Lustre and create a new tarball.

---

5. **Launch the test suite. Run:**

```
su - vxs0
. ../profile
tcc -e -a /mnt/lustre/TESTROOT -s scen.exec -p
```

16.3 Building and Running a POSIX Compliance Test Suite on Lustre

This section describes how to build and run a POSIX compliance test suite for a compiler and architecture for which we do not provide a quick start package.

16.3.1 Building the Test Suite from Scratch

This section describes building a POSIX compliance suite to test a Lustre file system.

1. **Download all POSIX files in** http://downloads.lustre.org/public/tools/benchmarks/posix
   - tet_vsxgen_3.02.tgz
   - lts_vsx-pcts2.0beta2.tgz
   - install.sh
   - myscen.bld
   - myscen.exec

Note – We now use the latest release of the LSB-VSX POSIX test suite (lts_vsx-pcts2.0beta2.tgz) and the generic TET/VSXgen framework (tet_vsxgen_3.02.tgz). In this release, the issue of "getgroups() did not return NGROUPS_MAX" has been fixed.

---

2. **DO NOT configure or mount a Lustre file system yet.**

3. **Run the** `{{{install.sh}}}` **script and select** `/home/tet` **for the root directory for the test suite installation. Say 'y' to install the users and groups. Accept the defaults to install the packages.**

4. **Create a temporary directory to hold the POSIX tests while they are being built. Run:**

   ```
   mkdir -p /mnt/lustre/TESTROOT;chown vsx0.vsxg0 !$
   ```

5. **Log in as the test user. Run:**

   ```
   su - vsx0
   ```

6. **Build the test suite. Run:**

   ```
   ../setup.sh
   ```

   Most of the default answers are correct, except the root directory from which to run the testsets. For this you should specify `/mnt/lustre/TESTROOT`. For "`Install pseudolanguages?`", answer 'n'.

7. **When the script prompts "Install scripts into TESTROOT/BIN..?", do not stop the script from running (this does not work). Instead, use another terminal to replace the existing files with the downloaded files. Enter:**

   ```
   cp .../myscen.bld  /home/tet/test_sets/scen.bld
   cp .../myscen.exec /home/tet/test_sets/scen.exec
   ```

   This confines the tests that are run to those relevant for file systems, avoiding hours of running other tests on sockets, math, stdio, libc, shell, etc.

8. **Continue with the installation at this point. Answer '**`y`**' to the "**`Build testsets`**" question.**

   The script builds and installs all file system tests and then runs them all. Although the script is running the files on a local file system, this is a valuable baseline for comparison with the behavior of Lustre.

   The results are put into `/home/tet/test_sets/results/0002e/journal`. It is suggested that you rename or symlink this directory to `/home/tet/test_sets/results/ext3/journal` (or the name of the local file system that the test was run on).

   Running the full test should only take about 5 minutes.

9. **Answer 'n' to re-running just the failed tests.**

   The results (in a table) are in `/home/tet/test_sets/results/report`.

10. **Save the test suite for later use, to run additional tests on a Lustre file system. Tar up the tests to avoid rebuilding them each time. Enter:**

```
tar cvzf TESTROOT.tgz -C /mnt/lustre TESTROOT
```

Tip – At this time, you probably want to remove the installed tests, to save a bit of space and, more importantly, to avoid confusion if you forget to mount your Lustre file system before running the tests.

16.3.2 Running the Test Suite Against Lustre

1. **As root, set up your Lustre file system, mounted on /mnt/lustre (e.g., `sh llmount.sh`) and untar the POSIX tests back to their home. Enter:**

```
tar --same-owner -xzpvf /path/to/tarball/TESTROOT.tgz -C
/mnt/lustre
```

2. **As the vsx0 user, you can re-run the tests as many times as necessary. If you are newly su'd or logged in as the vsx0 user, you need to source the environment with '. profile' so your path and other environment is set up correctly. To run the tests, enter:**

```
. /home/tet/profile
tcc -e -s scen.exec -a /mnt/lustre/TESTROOT -p
```

   Each new result is put in a new directory under `/home/tet/test_sets/results` and given a directory name similar to 0004e, an increasing number that ends with `e` for test execution or `b` for building the tests).

3. **To look at a formatted report, enter:**

```
vrpt results/0004e/journal | less
```

   Some tests are "Unsupported", "Untested", or "Not In Use", which does not necessarily indicate a problem.

4. **To compare two test results, run:**

   ```
   vrptm results/ext3/journal results/0004e/journal | less
   ```

   This is more interesting than looking at the result of a single test, since it helps find test failures that are specific to the file system instead of the Linux VFS or kernel. Up to 6 test results can be compared at one time.

   It is often useful to rename the results directory to have more meaningful names (such as `before_unlink_fix`).

16.4 Isolating and Debugging Failures

When failures occur, you need to gather information about what is happening at runtime. For example, some tests may cause kernel panics depending on your configuration.

- The POSIX compliance suite does not have debugging enabled by default, so it is useful to turn on the debugging options of VSX. Two important debug options reside in the `tetexec.cfg` configuration file, under the TESTROOT directory:
  - VSX_DBUG_FILE=output_file - If you are running the test under UML with hostfs support, use a file on the hostfs as the debug output file. In the case of a crash, the debug output is then be safely written to the debug file.

Note – The default value for this option puts the debug log under your test directory in /mnt/lustre/TESTROOT, which may not be useful if you experience a kernel panic and lustre (or your machine) crashes.

  - VSX_DBUG_FLAGS=xxxxx - For detailed information about debug flags, refer to the documentation included with the POSIX test suite. The following example causes VSX to output all debug messages:

    ```
    VSX_DBUG_FLAGS=t:d:n:f:F:L:l,2:p:P
    ```

- VSX is based on the TET framework which provides common libraries for VSX. You can have TET print verbose debug messages by inserting the -T option when running the tests:

  ```
  tcc -Tall5 -e -s scen.exec -a /mnt/lustre/TESTROOT -p 2>&1 | tee
  /tmp/POSIX-command-line-output.log
  ```

- VSX prints detailed messages in the report for failed tests. This includes the test strategy, the kind of operations done by the test suite, and what is going wrong.

Each subtest (e.g., 'access', 'create') usually contains a number of single tests. The report shows exactly which single test fails. In this case, you can find more information directly from the VSX source code. For example, if the fifth single test of subtest chmod failed, you could look at the source:

```
/home/tet/test_sets/tset/POSIX.os/files/chmod/chmod.c
```

...which contains a single test array:

```
public  struct tet_testlist tet_testlist[] = {
  test1, 1,
  test2, 2,
  test3, 3,
  test4, 4,
  test5, 5,
  test6, 6,
  test7, 7,
  test8, 8,
  test9, 9,
  test10, 10,
  test11, 11,
  test12, 12,
  test13, 13,
  test14, 14,
  test15, 15,
  test16, 16,
  test17, 17,
  test18, 18,
  test19, 19,
  test20, 20,
  test21, 21,
  test22, 22,
  test23, 23,
NULL, 0
};
```

If this single test is causing problems, as in the case of a kernel panic, or if you are trying to isolate a single failure, it may be useful to edit the `tet_testlist` array down to the single test in question and then recompile the test suite. Then, you can create a new tarball of the resulting TESTROOT directory, named appropriately (e.g, TESTROOT-chmod-5-only.tgz) and re-run the POSIX suite using the steps above.

It may also be helpful to edit the `scen.exec` file to run only the test set in question:

```
all
     "total tests in POSIX.os 1"
     /tset/POSIX.os/files/chmod/T.chmod
```

Note – Rebuilding individual POSIX tests is not straightforward due to the reliance on tcc. One option is to substitute edited source files into the source tree while following the manual installation procedure described above and let the existing POSIX install scripts do the work. The installation scripts (specifically `/home/tet/test_sets/run_testsets.sh`), contain relevant commands to build the test suite -- something akin to `tcc -p -b -s $HOME/scen.bld $*` -- but these commands may not work outside the scripts. Let us know if you get better mileage rebuilding these tests.

CHAPTER 17

Benchmarking

The benchmarking process involves identifying the highest standard of excellence and performance, learning and understanding these standards, and finally adapting and applying them to improve the performance. Benchmarks are most often used to provide an idea of how fast any software or hardware runs.

Complex interactions between I/O devices, caches, kernel daemons, and other OS components result in behavior that is difficult to analyze. Moreover, systems have different features and optimizations, so no single benchmark is always suitable. The variety of workloads that these systems experience also adds in to this difficulty. One of the most widely researched areas in storage subsystem is file system design, implementation, and performance.

This chapter describes benchmark suites to test Lustre and includes the following sections:

17.1 Bonnie++ Benchmark

Bonnie++ is a benchmark suite that having aim of performing a number of simple tests of hard drive and file system performance. Then you can decide which test is important and decide how to compare different systems after running it. Each Bonnie++ test gives a result of the amount of work done per second and the percentage of CPU time utilized.

There are two sections to the program's operations. The first is to test the I/O throughput in a fashion that is designed to simulate some types of database applications. The second is to test creation, reading, and deleting many small files in a fashion similar to the usage patterns.

Bonnie++ is a benchmark tool that test hard drive and file system performance by sequential I/O and random seeks. Bonnie++ tests file system activity that has been known to cause bottlenecks in I/O-intensive applications.

To install and run the Bonnie++ benchmark:

1. **Download the most recent version of the Bonnie++ software:**

   http://www.coker.com.au/bonnie++/

2. **Install and run the Bonnie++ software (per the ReadMe file accompanying the software).**

   Sample output:

```
Version  1.03 --Sequential Output-- --Sequential Input- --Random--
-Per Chr- --Block-- -Rewrite- -Per Chr- --Block-- --Seeks--
MachineSize K/sec %CP K/sec %CP K/sec %CP K/sec %CP K/sec %CP /sec
%CP
mds         2G        3811822   21245  10     51967  10  90.00

------Sequential Create------ --------Random Create--------

-Create-- --Read--- -Delete-- -Create-- --Read--- -Delete--
files  /sec %CP  /sec %CP  /sec %CP  /sec %CP  /sec %CP  /sec %CP
16   510   0 +++++ +++   283   1   465   0 +++++ +++   291   1
mds,2G,,,38118,22,21245,10,,,51967,10,90.0,0,16,510,0,+++++,+++,28
3,1,465,0,+++++,+++,291,1
```

```
Version  1.03 --Sequential Output-- --Sequential Input- --Random--
-Per Chr- --Block-- -Rewrite- -Per Chr- --Block-- --Seeks--
MachineSize K/sec %CP K/sec %CP K/sec %CP K/sec %CP K/sec %CP  /sec
%CP
mds              2G 27460  92 41450  25 21474  10 19673  60 52871
10  88.0   0

------Sequential Create------ --------Random Create--------

-Create-- --Read--- -Delete-- -Create-- --Read--- -Delete--

files  /sec %CP  /sec %CP  /sec %CP  /sec %CP  /sec %CP  /sec %CP
16 29681  99 +++++ +++ 30412  90 29568  99 +++++ +++ 28077  82
mds,2G,27460,92,41450,25,21474,10,19673,60,52871,10,88.0,0,16,2968
1,99,+++++,+++,30412,90,29568,99,+++++,+++,28077,82
```

17.2 IOR Benchmark

Use the IOR_Survey script to test the performance of the lustre file systems. It uses IOR (Interleaved or Random), a script used for testing performance of parallel file systems using various interfaces and access patterns. IOR uses MPI for process synchronization.

Under the control of compile-time defined constants (and, to a lesser extent, environment variables), I/O is done via MPI-IO. The data are written and read using independent parallel transfers of equal-sized blocks of contiguous bytes that cover the file with no gaps and that do not overlap each other. The test consists of creating a new file, writing it with data, then reading the data back.

The IOR benchmark, developed by LLNL, tests system performance by focusing on parallel/sequential read/write operations that are typical of scientific applications.

To install and run the IOR benchmark:

1. **Satisfy the prerequisites to run IOR.**

   a. **Download lam 7.0.6 (local area multi-computer):**
   http://www.lam-mpi.org/7.0/download.php

   b. **Obtain a Fortran compiler for the Fedora Core 4 operating system.**

   c. **Download the most recent version of the IOR software:**
   http://sourceforge.net/projects/ior-sio

2. **Install the IOR software (per the ReadMe file and User Guide accompanying the software).**

3. **Run the IOR software. In user mode, use the lamboot command to start the lam service and use appropriate Lustre-specific commands to run IOR (described in the IOR User Guide).**

   Sample Output:

```
IOR-2.9.0: MPI Coordinated Test of Parallel I/O
Run began: Fri Sep 29 11:43:56 2006
Command line used: ./IOR -w -r -k -O lustrestripecount 10 –o test
Machine: Linux mds
Summary:
api               = POSIX
test filename     = test
access            = single-shared-file
clients           = 1 (1 per node)
repetitions       = 1
xfersize          = 262144 bytes
blocksize         = 1 MiB
aggregate filesize= 1 MiB
access  bw(MiB/s) block(KiB)xfer(KiB) open(s)wr/rd(s)close(s)iter
------  --------- --------- --------  --------------------------
write   173.89    1024.00   256.00    0.0000300.0057010.0000160
read    278.49    1024.00   256.00    0.0000090.0035660.0000120

Max Write: 173.89 MiB/sec (182.33 MB/sec)
Max Read:  278.49 MiB/sec (292.02 MB/sec)

Run finished: Fri Sep 29 11:43:56 2006
```

17.3 IOzone Benchmark

IOZone is a file system benchmark tool which generates and measures a variety of file operations. Iozone has been ported to many machines and runs under many operating systems. Iozone is useful to perform a broad file system analysis of a vendor's computer platform. The benchmark tests file I/O performance for the operations like read, write, re-read, re-write, read backwards, read strided, fread, fwrite, random read/write, pread/pwrite variants, aio_read, aio_write, mm, etc.

The IOzone benchmark tests file I/O performance for the following operations: read, write, re-read, re-write, read backwards, read strided, fread, fwrite, random read/write, pread/pwrite variants, aio_read, aio_write, and mmap.

To install and run the IOzone benchmark:

1. **Download the most recent version of the IOZone software from this location:**

   http://www.iozone.org

2. **Install the IOZone software (per the ReadMe file accompanying the IOZone software).**

3. **Run the IOZone software (per the ReadMe file accompanied with the IOZone software).**

```
Sample Output
Iozone:    Performance Test of File I/O
           Version $Revision: 3.263 $
Compiled for 32 bit mode.
Build: linux
Contributors:William Norcott, Don Capps, Isom Crawford,
           Kirby Collins, Al Slater, Scott Rhine, Mike Wisner,
           Ken Goss, Steve Landherr, Brad Smith, Mark Kelly,
           Dr. Alain CYR, Randy Dunlap, Mark Montague, Dan Million,
           Jean-Marc Zucconi, Jeff Blomberg, Erik Habbinga,
           Kris Strecker, Walter Wong.
Run began: Fri Sep 29 15:37:07 2006
Network distribution mode enabled.
Command line used: ./iozone -+m test.txt
Output is in Kbytes/sec
Time Resolution = 0.000001 seconds.
Processor cache size set to 1024 Kbytes.
Processor cache line size set to 32 bytes.
File stride size set to 17 * record size.
random     random    bkwd   record stride KB     reclen    write
rewrite    read      reread read   write  read   rewrite   read
fwrite     frewrite  fread  freread

512        4         194309 406651 728276 792701 715002    498592
638351     700365    587235 190554 378448 686267 765201

iozone test complete.
```

CHAPTER 18

Lustre I/O Kit

This chapter describes the Lustre I/O kit and PIOS performance tool, and includes the following sections:

18.1 Lustre I/O Kit Description and Prerequisites

The Lustre I/O kit is a collection of benchmark tools for a Lustre cluster. The I/O kit can be used to validate the performance of the various hardware and software layers in the cluster and also as a way to find and troubleshoot I/O issues.

The I/O kit contains three tests. The first surveys basic performance of the device and bypasses the kernel block device layers, buffer cache and file system. The subsequent tests survey progressively higher layers of the Lustre stack. Typically with these tests, Lustre should deliver 85-90% of the raw device performance.

It is very important to establish performance from the “bottom up” perspective. First, the performance of a single raw device should be verified. Once this is complete, verify that performance is stable within a larger number of devices. Frequently, while troubleshooting such performance issues, we find that array performance with all LUNs loaded does not always match the performance of a single LUN when tested in isolation. After the raw performance has been established, other software layers can be added and tested in an incremental manner.

18.1.1 Downloading an I/O Kit

You can download the I/O kit from:

http://downloads.clusterfs.com/public/tools/lustre-iokit/

In this directory, you will find two packages:

- `lustre-iokit` consists of a set of developed and supported by the Lustre group.
- `scali-lustre-iokit` is a Python tool maintained by Scali team, and is not discussed in this manual.

18.1.2 Prerequisites to Using an I/O Kit

The following prerequisites must be met to use the Lustre I/O kit:

- password-free remote access to nodes in the system (normally obtained via ssh or rsh)
- Lustre file system software
- `sg3_utils` for the `sgp_dd` utility

18.2 Running I/O Kit Tests

As mentioned above, the I/O kit contains these test tools:

- `sgpdd_survey`
- `obdfilter_survey`
- `ost_survey`

18.2.1 sgpdd_survey

Use the `sgpdd_survey` tool to test bare metal performance, while bypassing as much of the kernel as possible. This script requires the `sgp_dd` package, although it does not require Lustre software. This survey may be used to characterize the performance of a SCSI device by simulating an OST serving multiple stripe files. The data gathered by this survey can help set expectations for the performance of a Lustre OST exporting the device.

The script uses `sgp_dd` to carry out raw sequential disk I/O. It runs with variable numbers of `sgp_dd` threads to show how performance varies with different request queue depths.

The script spawns variable numbers of `sgp_dd` instances, each reading or writing a separate area of the disk to demonstrate performance variance within a number of concurrent stripe files.

The device(s) used must meet one of the two tests described below:

SCSI device:

Must appear in the output of `sg_map` (make sure the kernel module "sg" is loaded)

Raw device:

Must appear in the output of `raw -qa`

If you need to create raw devices in order to use the sgpdd_survey tool, note that raw device 0 cannot be used due to a bug in certain versions of the "raw" utility (including that shipped with RHEL4U4.)

You may not mix raw and SCSI devices in the test specification.

Caution – The `sgpdd_survey` script overwrites the device being tested, which results in the LOSS OF ALL DATA on that device. Exercise caution when selecting the device to be tested.

The sgpdd_survey script must be customized according to the particular device being tested and also according to the location where it should keep its working files. Customization variables are described explicitly at the start of the script.

When the sgpdd_survey script runs, it creates a number of working files and a pair of result files. All files start with the prefix given by the script variable ${rslt}.

```
${rslt}_<date/time>.summary same as stdout
${rslt}_<date/time>_* tmp files
${rslt}_<date/time>.detail collected tmp files for post-mortem
```

The summary file and stdout should contain lines like this:

```
total_size 8388608K rsz 1024 thr 1 crg 1 180.45 MB/s 1 x 180.50 \
=/ 180.50 MB/s
```

The number immediately before the first MB/s is bandwidth, computed by measuring total data and elapsed time. The remaining numbers are a check on the bandwidths reported by the individual sgp_dd instances.

If there are so many threads that the sgp_dd script is unlikely to be able to allocate I/O buffers, then "ENOMEM" is printed.

If one or more sgp_dd instances do not successfully report a bandwidth number, then "failed" is printed.

18.2.2 obdfilter_survey

The `obdfilter_survey` script processes sequential I/O with varying numbers of threads and objects (files) by using `lctl` to drive the `echo_client` connected to local or remote obdfilter instances or remote obdecho instances. It can be used to characterize the performance of the following Lustre components:

OSTs

The script exercises one or more instances of obdfilter directly. The script may run on one or more nodes, for example, when the nodes are all attached to the same multi-ported disk subsystem.

Tell the script the names of all obdfilter instances (which should be up and running already). If some instances are on different nodes, specify their hostnames too (for example, `node1:ost1`). Alternately, you can pass parameter `case=disk` to the script. (The script automatically detects the local obdfilter instances.)

All obdfilter instances are driven directly. The script automatically loads the obdecho module (if required) and creates one instance of `echo_client` for each obdfilter instance.

Network

The script drives one or more instances of the obdecho server via instances of `echo_client` running on one or more nodes. Pass the parameters `case=network` and `target=''<hostname/ip_of_server>''` to the script. For each nework case, the script does the required setup.

Striped File System Over the Network

The script drives one or more instances of obdfilter via instances of `echo_client` running on one or more nodes.

Tell the script the names of the OSCs (which should be up and running). Alternately, you can pass the parameter `case=netdisk` to the script. The script will use all of the local OSCs.

Note – The `obdfilter_survey` script is NOT scalable to 100s of nodes since it is only intended to measure individual servers, not the scalability of the entire system.

Note – The `obdfilter_survey` script must be customized, depending on the components under test and where the script's working files should be kept. Customization variables are clearly described in the script (Customization Variables section). In particular, refer to the maximum supported value ranges for customization variables.

18.2.2.1 Running obdfilter_survey Against a Local Disk

The obdfilter_survey script supports **automatic** and **manual** runs against a local disk. Obdfilter-survey profiles the overall throughput of storage hardware[1], by sending ranges of workloads to the OSTs (varied in thread counts and I/O sizes).

When the obdfilter_survey script is complete, it provides information on the performance abilities of the storage hardware and shows the saturation points. If you use plot scripts on the data, this information is shown graphically.

To run the obdfilter_survey script, create a normal Lustre configuration; no special setup is needed.

To perform an automatic run:

1. **Set up the Lustre file system.**
2. **Verify that the obdecho.ko module is present.**
3. **Run the obdfilter_survey script with the parameter case=disk. For example:**

```
$ nobjhi=2 thrhi=2 size=1024 case=disk sh obdfilter-survey
```

To perform a manual run:

1. **List all OSTs you want to test. (You do not have to specify an MDS or LOV.)**
2. **On all OSSs, run:**

```
$ mkfs.lustre --fsname spfs --mdt --mgs /dev/sda
```

Caution – Write tests are destructive. This test should be run before the Lustre file system is started. If you do this, you will not need to reformat to restart Lustre system. However, if the obdfilter_survey test is terminated before it completes, you may have to remove objects from the disk.

1. The sgpdd-survey profiles individual disks. This script is destructive, and should not be run anywhere you want to preserve existing data.

3. **Determine the obdfilter instance names on all Lustre clients. The device names appear in the fourth column of the `lctl dl` command output. For example:**

```
$ pdsh -w oss[01-02] lctl dl |grep obdfilter |sort
oss01:      0 UP obdfilter oss01-sdb oss01-sdb_UUID 3
oss01:      2 UP obdfilter oss01-sdd oss01-sdd_UUID 3
oss02:      0 UP obdfilter oss02-sdi oss02-sdi_UUID 3
...
```

In this example, the obdfilter instance names are `oss01-sdb`, `oss01-sdd`, and `oss02-sdi`. Since you are driving obdfilter instances directly, set the shell array variable, targets, to the names of the obdfilter instances. For example:

```
targets='oss01:oss01-sdb oss01:oss01-sdd oss02:oss02-sdi'\
./obdfilter-survey
```

18.2.2.2 Running obdfilter_survey Against a Network

The obdfilter_survey script can only be run automatically against a network; no manual test is supported.

To run the network test, a specific Lustre setup is needed. Make sure that these configuration requirements have been met.

- Install all Lustre modules, including obdecho.
- Start lctl and check the device list, which must be empty.
- Use a password-less entry between the client and server machines, to avoid having to type the password.

To perform an automatic run:

1. **Run the obdfilter_survey script with the parameters case=netdisk and targets= "<hostname/ip_of_server>". For example:**

```
$ nobjhi=2 thrhi=2 size=1024 targets="<hostname/ip_of_server>" \
case=network sh obdfilter-survey
```

On the server side, you can see the statistics at:

```
/proc/fs/lustre/obdecho/<echo_srv>/stats
```

where 'echo_srv' is the obdecho server created by the script.

18.2.2.3 Running obdfilter_survey Against a Network Disk

The obdfilter_survey script can be run automatically or manually against a network disk.

To run the network disk test, create a Lustre configuration using normal methods; no special setup is needed.

To perform an automatic run:

1. **Set up the Lustre file system with the required OSTs.**
2. **Verify that the obdecho.ko module is present.**
3. **Run the obdfilter_survey script with the parameter case=netdisk. For example:**

```
$ nobjhi=2 thrhi=2 size=1024 case=netdisk sh obdfilter-survey
```

To perform a manual run:

1. **Run the obdfilter_survey script and tell the script the names of all echo_client instances (which should be up and running already).**

```
$ nobjhi=2 thrhi=2 size=1024 targets="<osc_name> ..." \ sh
obdfilter-survey
```

18.2.2.4 Output Files

When the obdfilter_survey script runs, it creates a number of working files and a pair of result files. All files start with the prefix given by ${rslt}.

File	Description
${rslt}.summary	Same as stdout
${rslt}.script_*	Per-host test script files
${rslt}.detail_tmp*	Per-OST result files
${rslt}.detail	Collected result files for post-mortem

The obdfilter_survey script iterates over the given number of threads and objects performing the specified tests and checks that all test processes have completed successfully.

Note – The `obdfilter_survey` script may not clean up properly if it is aborted or if it encounters an unrecoverable error. In this case, a manual cleanup may be required, possibly including killing any running instances of 'lctl' (local or remote), removing echo_client instances created by the script and unloading obdecho.

18.2.2.5 Script Output

The summary file and stdout of the obdfilter_survey script contain lines such as:

```
ost 8 sz 67108864K rsz 1024 obj 8 thr 8 write 613.54 [ 64.00, 82.00]
```

Where:

Variable	Supported Type
ost8	Total number of OSTs being tested.
sz 67108864K	Total amount of data read or written (in KB).
rsz 1024	Record size (size of each echo_client I/O, in KB).
obj 8	Total number of objects over all OSTs.
thr 8	Total number of threads over all OSTs and objects.
write	Test name. If more tests have been specified, they all appear on the same line.
613.54	Aggregate bandwidth over all OSTs (measured by dividing the total number of MB by the elapsed time).
[64, 82.00]	Minimum and maximum instantaneous bandwidths on an individual OST.

Note – Although the numbers of threads and objects are specified per-OST in the customization section of the script, the reported results are aggregated over all OSTs.

18.2.2.6 Visualizing Results

It is useful to import the obdfilter_survey script summary data (it is fixed width) into Excel (or any graphing package) and graph the bandwidth versus the number of threads for varying numbers of concurrent regions. This shows how the OSS performs for a given number of concurrently-accessed objects (files) with varying numbers of I/Os in flight.

It is also extremely useful to record average disk I/O sizes during each test. These numbers help locate pathologies in the system when the file system block allocator and the block device elevator.

The `plot-obdfilter` script (included) is an example of processing output files to a `.csv` format and plotting a graph using gnuplot.

18.2.3 ost_survey

The `ost_survey` tool is a shell script that uses `lfs setstripe` to perform I/O against a single OST. The script writes a file (currently using dd) to each OST in the Lustre file system, and compares read and write speeds. The `ost_survey` tool is used to detect misbehaving disk subsystems.

Note – We have frequently discovered wide performance variations across all LUNs in a cluster.

To run the `ost_survey` script, supply a file size (in KB) and the Lustre mount point. For example, run:

```
$ ./ost-survey.sh 10 /mnt/lustre
Average read Speed:        6.73
Average write Speed:       5.41
read - Worst OST indx 0    5.84 MB/s
write - Worst OST indx 0   3.77 MB/s
read - Best OST indx 1     7.38 MB/s
write - Best OST indx 1    6.31 MB/s
3 OST devices found
Ost index 0 Read speed 5.84   Write speed  3.77
Ost index 0 Read time  0.17   Write time   0.27
Ost index 1 Read speed 7.38   Write speed  6.31
Ost index 1 Read time  0.14   Write time   0.16
Ost index 2 Read speed 6.98   Write speed  6.16
Ost index 2 Read time  0.14   Write time   0.16
```

18.3 PIOS Test Tool

The PIOS test tool is a parallel I/O simulator for Linux and Solaris. PIOS generates I/O on file systems, block devices and zpools similar to what can be expected from a large Lustre OSS server when handling the load from many clients. The program generates and executes the I/O load in a manner substantially similar to an OSS, that is, multiple threads take work items from a simulated request queue. It forks a CPU load generator to simulate running on a system with additional load.

PIOS can read/write data to a single shared file or multiple files (default is a single file). To specify multiple files, use the --fpp option. (It is better to measure with both single and multiple files.) If the final argument is a file, block device or zpool, PIOS writes to `RegionCount` regions in one file. PIOS issues I/O commands of size `ChunkSize`. The regions are spaced apart Offset bytes (or, in the case of many files, the region starts at `Offset` bytes). In each region, `RegionSize` bytes are written or read, one `ChunkSize` I/O at a time. Note that:

`ChunkSize <= Regionsize <= Offset`

Multiple runs can be specified with comma separated lists of values for `ChunkSize`, `Offset`, `RegionCount`, `ThreadCount`, and `RegionSize`. Multiple runs can also be specified by giving a starting (low) value, increase (in percent) and high value for each of these arguments. If a low value is given, no value list or value may be supplied.

Every run is given a timestamp, and the timestamp and offset are written with every chunk (to allow verification). Before every run, PIOS executes the pre-run shell command. After every run, PIOS executes the post-run command. Typically, this is used to clear and collect statistics for the run, or to start and stop statistics gathering during the run. The timestamp is passed to both pre-run and post-run.

For convenience, PIOS understands byte specifiers and uses:

K,k for kilobytes (2<<10)

M,m for megabytes (2<<20)

G,g for gigabytes (2<<30)

T,t for terabytes (2<<40)

Download the PIOS test tool at:

http://downloads.clusterfs.com/public/tools/benchmarks/pios/

18.3.1 Synopsis

```
pios
[--chunksize|-c =values, (--chunksize_low|-a =value
--chunksize_high|-b =value --chunksize_incr|-g =value)]

[--offset|-o =values, (--offset_low|-m =value --offset_high|-q =value
--offset_incr|-r =value)]

[--regioncount|-n =values, (--regioncount_low|-i =value
--regioncount_high|-j =value --regioncount_incr|-k =value)]

[--threadcount|-t =values, (--threadcount_low|-l =value
--threadcount_high|-h =value --threadcount_incr|-e =value)]

[--regionsize|-s =values, (--regionsize_low|-A =value
--regionsize_high|-B =value --regionsize_incr|-C =value)]

[--directio|-d, --posixio|-x, --cowio|-w} [--cleanup|-L
--threaddelay|-T =ms --regionnoise|-I ==shift
--chunknoise|-N =bytes -fpp|-F ]

[--verify|-V =values]

[--prerun|-P =pre-command --postrun|-R =post-command]

[--path|-p =output-file-path]
```

18.3.2 PIOS I/O Modes

There are several supported PIOS I/O modes:

POSIX I/O:

This is the default operational mode where I/O is done using standard POSIX calls, such as pwrite/pread. This mode is valid on both Linux and Solaris.

DIRECT I/O:

This mode corresponds to the O_DIRECT flag in open(2) system call, and it is currently applicable only to Linux. Use this mode when using PIOS on the ldiskfs file system on an OSS.

COW I/O:

This mode corresponds to the copy overwrite operation where file system blocks that are being overwritten were copied to shadow files. Only use this mode if you want to see overhead of preserving existing data (in case of overwrite). This mode is valid on both Linux and Solaris.

18.3.3 PIOS Parameters

PIOS has five basic parameters to determine the amount of data that is being written.

ChunkSize(c):

Amount of data that a thread writes in one attempt. `ChunkSize` should be a multiple of file system block size.

RegionSize(s):

Amount of data required to fill up a region. PIOS writes a chunksize of data continuously until it fills the regionsize. `RegionSize` should be a multiple of `ChunkSize`.

RegionCount(n):

Number of regions to write in one or multiple files. The total amount of data written by PIOS is `RegionSize` x `RegionCount`.

ThreadCount(t):

Number of threads working on regions.

Offset(o):

Distance between two successive regions when all threads are writing to the same file. In the case of multiple files, threads start writing in files at `Offset` bytes.

Parameter	Description
--chunknoise = N	N is a byte specifier. When performing an I/O task, add a random signed integer in the range [-N,N] to the chunksize. All regions are still fully written. This randomizes the I/O size to some extent.
--chunksize = N[,N2,N3...]	N is a byte specifier and performs I/O in chunks of N kilo-, mega-, giga- or terabyte. You can give a comma separated list of multiple values. This argument is mutually exclusive with --chunksize_low. Note that each thread allocates a buffer of size chunksize + chunknoise for use during the run.
--chunksize_low=L **--chunksize_high=H** **--chunksize_incr=F**	Performs a sequence of operations starting with a chunksize of L, increasing it by F% each time until chunksize exceeds H.
--cleanup	Removes files that were created during the run. If there is an encounter for existing files, they are over-written.
--directio **--posixio** **--cowio**	One of these arguments must be passed to indicate if DIRECT I/O, POSIX I/O or COW I/O is used.
--offset=O[,O2,O3...]	The argument is a byte specifier or a list of specifiers. Each run uses regions at offset multiple of O in a single file. If the run targets multiple files, then the I/O writes at offset O in each file.
--offset_low=OL **--offset_high=OH** **--offset_inc=PH**	The arguments are byte specifiers. They generate runs with a range of offsets starting at OL, increasing P% until the region size exceeds OH. Each of these arguments is exclusive with the offset argument.
--prerun="pre-command"	Before each run, executes the pre-command as a shell command through the system(3) call. The timestamp of the run is appended as the last argument to the pre-command string. Typically, this is used to clear statistics or start a data collection script when the run starts.
--postrun="post-command"	After each run, executes the post-command as a shell command through the system(3) call. The timestamp of the run is appended as the last argument to the pre-command string. Typically, this is used to append statistics for the run or close an open data collection script when the run completes.

Parameter	Description
--regioncount=N[,N2,N3...]	PIOS writes to N regions in a single file or block device or to N files.
--regioncount_low=RL **--regioncount_high=RH** **--regioncount_inc=P**	Generate runs with a range of region counts starting at TL, increasing P% until the thread count exceeds RH. Each of these arguments is exclusive with the regioncount argument.
--regionnoise=k	When generating the next I/O task, do not select the next chunk in the next stream, but shift a random number with a maximum noise of shifting k regions ahead. The run will complete when all regions are fully written or read. This merely introduces a randomization of the ordering.
--regionsize=S[,S2,S3...]	The argument is a byte specifier or a list of byte specifiers. During the run(s), write S bytes to each region.
--regionsize_low=RL **--regionsize_high=RH** **--regionsize_inc=P**	The arguments are byte specifiers. Generate runs with a range of region sizes starting at TL, increasing P% until the region size exceeds RH. Each argument is exclusive with the regionsize argument.
--threadcount=T[,T2,T3...]	PIOS runs with T threads performing I/O. A sequence of values may be given.
--threadcount_low=TL **--threadcount_high=TH** **--threadcount_inc=TP**	Generate runs with a range of thread counts starting at TL, increasing TP% until the thread count exceeds TH. Each of these arguments is exclusive with the threadcount argument.
--threaddelay=ms	A random amount of noise not exceeding ms is inserted between the time that a thread identifies as the next chunk it needs to read or write and the time it starts the I/O.
--fpp	Where threads write to files: • **fpp** indicates files per process behavior where threads write to multiple files. • **sff** indicates single shared files where all threads write to the same file.
--verify-V=timestamp [,timestamp2,timestamp3]|--verify|-V	Verify a written file or set of files. A single timestamp or sequence of timestamps can be given for each run, respectively. If no argument is passed, the verification is done from timestamps read from the first location of files previously written in the test. If sequence is given, then each run verifies the timestamp accordingly. If a single timestamp is given, then it is verified with all files written.

18.3.4 PIOS Examples

To create a 1 GB load with a different number of threads:

In one file:

```
pios -t 1,2,4, 8,16, 32, 64, 128 -n 128 -c 1M -s 8M -o 8M \
--load=posixio -p /mnt/lustre
```

In multiple files:

```
pios -t 1,2,4, 8,16, 32, 64, 128 -n 128 -c 1M -s 8M -o 8M \
--load=posixio,fpp -p /mnt/lustre
```

To create a 1 GB load with a different number of chunksizes on ldiskfs with direct I/O:

In one file:

```
pios -t 32 -n 128 -c 128K, 256K, 512K, 1M, 2M, 4M -s 8M -o 8M \
--load=directio -p /mnt/lustre
```

In multiple files:

```
pios -t 32 -n 128 -c 128K, 256K, 512K, 1M, 2M, 4M -s 8M -o 8M \
--load=directio,fpp -p /mnt/lustre
```

To create a 32 MB to 128 MB load with different RegionSizes on a Solaris zpool:

In one file:

```
pios -t 8 -n 16 -c 1M -A 2M -B 8M -C 100 -o 8M --load=posixio -p \
/myzpool/
```

In multiple files:

```
pios -t 8 -n 16 -c 1M -A 2M -B 8M -C 100 -o 8M --load=posixio, \
fpp -p /myzpool/
```

To read and verify timestamps:

Create a load with PIOS:

```
pios -t 40 -n 1024 -c 256K -s 4M -o 8M --load=posixio -p
/mnt/lustre
```

Keep the same parameters to read:

```
pios -t 40 -n 1024 -c 256K -s 4M -o 8M --load=posixio -p \
/mnt/lustre --verify
```

18.4 LNET Self-Test

LNET self-test helps site administrators confirm that Lustre Networking (LNET) has been properly installed and configured, and that underlying network software and hardware are performing according to expectations.

LNET self-test is a kernel module that runs over LNET and LNDs. It is designed to:

- Test the connection ability of the Lustre network
- Run regression tests of the Lustre network
- Test performance of the Lustre network

Note – Apart from the performance impact, LNET self-test is invisible to Lustre.

18.4.1 Basic Concepts of LNET Self-Test

This section describes basic concepts of LNET self-test, utilities and a sample script.

18.4.1.1 Modules

To run LNET self-test, these modules must be loaded: libcfs, lnet, lnet_selftest and one of the klnds (i.e, ksocklnd, ko2iblnd...). To load all necessary modules, run `modprobe lnet_selftest` (recursively loads the modules on which LNET self-test depends.

The LNET self-test cluster has two types of nodes:

- **Console node** - A single node that controls and monitors the test cluster. It can be any node in the test cluster.
- **Test nodes** - The nodes that run tests. Test nodes are controlled by the user via the console node; the user does not need to log into them directly.

The console and test nodes require all previously-listed modules to be loaded. (The userspace test node does not require these modules.)

Note – Test nodes can be in either kernel or userspace. A console user can invite a kernel test node to join the test session by running `lst add_group NID`, but the user cannot actively add a userspace test node to the test-session. However, the console user can passively accept a test node to the test session while the test node runs `lstclient` to connect to the console.

18.4.1.2 Utilities

LNET self-test has two user utilities, lst and lstclient.

- **lst** - The user interface for the self-test console (run on the console node). It provides a list of commands to control the entire test system, such as create session, create test groups, etc.
- **lstclient** - The userspace LNET self-test program (run on a test node). lstclient is linked with userspace LNDs and LNET. lstclient is not needed if a user just wants to use kernel space LNET and LNDs.

18.4.1.3 Session

In the context of LNET self-test, a session is a test node that can be associated with only one session at a time, to ensure that the session has exclusive use. Almost all operations should be performed in a session context. From the console node, a user can only operate nodes in his own session. If a session ends, the session context in all test nodes is destroyed.

The console node can be used to create, change or destroy a session (`new_session`, `end_session`, `show_session`). For more information, see Session.

18.4.1.4 Console

The console node is the user interface of the LNET self-test system, and can be any node in the test cluster. All self-test commands are entered from the console node. From the console node, a user can control and monitor the status of the entire test cluster (session). The console node is exclusive, meaning that a user cannot control two different sessions (LNET self-test clusters) on one node.

18.4.1.5 Group

An LNET self-test group is just a named collection of nodes. There are no restrictions on group membership, i.e., a node can be included in any number of groups, and any number of groups can exist in a single LNET self-test session.

Each node in a group has a rank, determined by the order in which it was added to the group, which is used to establish test traffic patterns.

A user can only control nodes in his/her session. To allocate nodes to the session, the user needs to add nodes to a group (of the session). All nodes in a group can be referenced by group's name. A node can be allocated to multiple groups of a session.

Note – A console user can associate kernel space test nodes with the session by running `lst add_group NIDs`, but a userspace test node cannot be actively added to the session. However, the console user can passively "accept" a test node to associate with test session while the test node running `lstclient` connects to the console node, i.e: `lstclient --sesid CONSOLE_NID --group NAME`).

18.4.1.6 Test

A test generates network load between two arbitrary groups of nodes - the test's "from" and "to" groups. When a test is running, each node in the "from" group sends requests to nodes in the "to" group, and receive responses in return. This activity is designed to mimic Lustre RPC traffic, i.e. the "from" group acts like a set of clients and the "to" group acts like a set of servers.

The traffic pattern and test intensity is determined several properties, including test type, distribution of test nodes, concurrency of test, RDMA operation type, etc. Several of the available test parameters are described below.

- **Type**: The test type determines the message pattern for a single request/response. Supported types are:
  - **Ping**: Small request / small response. Pings only generate small messages. They are useful to determine latency and small message overhead, and to simulate Lustre metadata traffic.
  - **brw**: Small request / bulk / small response. Brws include an additional phase where bulk data is either fetched from the request sender (brw write) or sent back to it (brw read) before the response is returned. The size of the bulk transfer is a test parameter. Brw tests are useful to determine network bandwidth and to simulate Lustre I/O traffic.

- **Distribution**: Determines which nodes in the "to" group communicate with each node in the "from" group. It allows you to specify a wide range of topologies, including one-to-one and all-to-all. Distribution divides the "from" group into subsets, which are paired with equivalent subsets from the "to" group so only nodes in matching subsets communicate. For example:

  --distribute 1:1 This is the default setting. Each "from" node communicates with the same rank (modules "to" group size) "to" node. Note that if there are more "from" nodes than "to" nodes, some "from" nodes may share the same "to" nodes. Also, if there are more "to" nodes than "from" nodes, some higher-ranked "to" nodes will be idle.

  --distribute 1:n (where 'n' is the size of the "to" group). Each "from" node communicates with every node in the "to" group.

- **Concurrency**: Determines how many requests each "from" node in a test keeps on the wire.

18.4.1.7 Batch

A batch is an arbitrary collection of tests which are started and stopped together; they run in parallel. Each test should belong to a batch; tests should not exist individually. Users can control a test batch (run, stop); they cannot control individual tests. Tests in a batch are non-destructive to the file system, and can be run in a normal Lustre environment (provided the performance impact is acceptable).

The simplest batch might contain only a single test - running brw to determine whether network bandwidth will be an I/O bottleneck. In this example, the "to" group is comprised of Lustre OSSes and the "from" group includes the compute nodes. Adding an second test to perform pings from a login node to the MDS could tell you how much checkpointing would affect the `ls -l` process.

18.4.1.8 Sample Script

These are the steps to run a sample LNET self-test script simulating the traffic pattern of a set of Lustre servers on a TCP network, accessed by Lustre clients on an InfiniBand network (connected via LNET routers). In this example, half the clients are reading and half the clients are writing.

1. **Load libcfs.ko, lnet.ko, ksocklnd.ko and lnet_selftest.ko on all test nodes and the console node.**

2. **Run this script on the console node:**

```
#!/bin/bash
export LST_SESSION=$$
lst new_session read/write
lst add_group servers 192.168.10.[8,10,12-16]@tcp
lst add_group readers 192.168.1.[1-253/2]@o2ib
lst add_group writers 192.168.1.[2-254/2]@o2ib
lst add_batch bulk_rw
lst add_test --batch bulk_rw --from readers --to servers \
brw read check=simple size=1M
lst add_test --batch bulk_rw --from writers --to servers \
brw write check=full size=4K
# start running
lst run bulk_rw
# display server stats for 30 seconds
lst stat servers & sleep 30; kill $!
# tear down
lst end_session
```

Note – This script can be easily adapted to pass the group NIDs by shell variables or command line arguments (making it good for general-purpose use).

18.4.2 LNET Self-Test Commands

The LNET self-test (lst) utility is used to issue LNET self-test commands. The lst utility takes a number of command line arguments. The first argument is the command name and subsequent arguments are command-specific.

18.4.2.1 Session

This section lists lst session commands.

Process Environment (LST_SESSION)

The `lst` utility uses the LST_SESSION environmental variable to identify the session locally on the self-test console node. This should be a numeric value that uniquely identifies all session processes on the node. It is convenient to set this to the process ID of the shell both for interactive use and in shell scripts. Almost all `lst` commands require LST_SESSION to be set.

new_session [--timeout SECONDS] [--force] NAME

Creates a new session.

–-timeout SECONDS	Console timeout value of the session. The session ends automatically if it remains idle (i.e., no commands are issued) for this period.
--force	Ends conflicting sessions. This determines who "wins" when one session conflicts with another. For example, if there is already an active session on this node, then this attempt to create a new session fails unless the `-force` flag is specified. However, if the `-force` flag is specified, then the other session is ended. Similarly, if this session attempts to add a node that is already "owned" by another session, the `-force` flag allows this session to "steal" the node.

name	A human-readable string to print when listing sessions or reporting session conflicts.
	
	`$ export LST_SESSION=$$`
	`$ lst new_session --force liangzhen`

end_session

Stops all operations and tests in the current session and clears the session's status.

```
$ lst end_session
```

show_session

Shows the session information. This command prints information about the current session. It does not require LST_SESSION to be defined in the process environment.

```
$ lst show_session
```

18.4.2.2 Group

This section lists lst group commands.

add_group NAME NIDs [NIDs...]

Creates the group and adds a list of test nodes to the group.

NAME	Name of the group.
NIDs	A string that may be expanded into one or more LNET NIDs.

```
$ lst add_group servers 192.168.10.[35,40-45]@tcp
$ lst add_group clients 192.168.1.[10-100]@tcp 192.168.[2,4].\
[10-20]@tcp
```

update_group NAME [--refresh] [--clean STATE] [--remove NIDs]

Updates the state of nodes in a group or adjusts a group's membership. This command is useful if some nodes have crashed and should be excluded from the group.

–-refresh	Refreshes the state of all inactive nodes in the group.	
–-clean STATUS	Removes nodes with a specified status from the group. Status may be:	
	active	The node is in the current session.
	busy	The node is now owned by another session.
	down	The node has been marked down.

	unknown The node's status has yet to be determined.
	invalid Any state but active.
--remove NIDs	Removes specified nodes from the group.

```
$ lst update_group clients --refresh
$ lst update_group clients --clean busy
$ lst update_group clients --clean invalid // \
invalid == busy || down || unknown
$ lst update_group clients --remove 192.168.1.[10-20]@tcp
```

list_group [NAME] [--active] [--busy] [--down] [--unknown] [--all]

Prints information about a group or lists all groups in the current session if no group is specified.

NAME	The name of the group.
--active	Lists the active nodes.
--busy	Lists the busy nodes.
--down	Lists the down nodes.
--unknown	Lists unknown nodes.
--all	Lists all nodes.

```
$ lst list_group
1) clients
2) servers
Total 2 groups
$ lst list_group clients
ACTIVE BUSY DOWN UNKNOWN TOTAL
3 1 2 0 6
$ lst list_group clients --all
192.168.1.10@tcp Active
192.168.1.11@tcp Active
192.168.1.12@tcp Busy
192.168.1.13@tcp Active
192.168.1.14@tcp DOWN
192.168.1.15@tcp DOWN
Total 6 nodes
$ lst list_group clients --busy
192.168.1.12@tcp Busy
Total 1 node
```

del_group NAME

Removes a group from the session. If the group is referred to by any test, then the operation fails. If nodes in the group are referred to only by this group, then they are kicked out from the current session; otherwise, they are still in the current session.

```
$ lst del_group clients
```

Userland client (lstclient --sesid NID --group NAME)

Use lstclient to run the userland self-test client. lstclient should be executed after creating a session on the console. There are only two mandatory options for lstclient:

–-sesid NID	The first console's NID.
–-group NAME	The test group to join.

```
Console $ lst new_session testsession
Client1 $ lstclient --sesid 192.168.1.52@tcp --group clients
```

Also, lstclient has a mandatory option that enforces LNET to behave as a server (start acceptor if the underlying NID needs it, use privileged ports, etc.):

--server_mode

For example:

```
Client1 $ lstclient --sesid 192.168.1.52@tcp |--group clients --server_mode
```

Note – Only the super user is allowed to use the `--server_mode` option.

18.4.2.3 Batch and Test

This section lists lst batch and test commands.

add_batch NAME

The default batch (named "batch") is created when the session is started. However, the user can specify a batch name by using `add_batch`:

```
$ lst add_batch bulkperf
```

add_test --batch BATCH [--loop #] [--concurrency #] [--distribute #:#] from GROUP --to GROUP TEST ...

Adds a test to batch. For now, TEST can be **brw** and **ping**:

–-loop #	Loop count of the test.
–-concurrency #	Concurrency of the test.
–-from GROUP	The source group (test client).
–-to GROUP	The target group (test server).

–-distribute #:#	The distribution of nodes in clients and servers. The first number of **distribute** is a subset of client (count of nodes in the "from" group). The second number of **distribute** is a subset of server (count of nodes in the "to" group); only nodes in two correlative subsets will talk. The following examples are illustrative:
	
	`Clients: (C1, C2, C3, C4, C5, C6)`
	`Server: (S1, S2, S3)`
	`--distribute 1:1`
	`(C1->S1), (C2->S2), (C3->S3), (C4->S1), (C5->S2), (C6->S3)`
	`\ /* -> means test conversation */`
	`--distribute 2:1`
	`(C1,C2->S1), (C3,C4->S2), (C5,C6->S3)`
	`--distribute 3:1`
	`(C1,C2,C3->S1), (C4,C5,C6->S2), (NULL->S3)`
	`--distribute 3:2`
	`(C1,C2,C3->S1,S2), (C4,C5,C6->S3,S1)`
	`--distribute 4:1`
	`(C1,C2,C3,C4->S1), (C5,C6->S2), (NULL->S3)`
	`--distribute 4:2`
	`(C1,C2,C3,C4->S1,S2), (C5, C6->S3, S1)`
	`--distribute 6:3`
	`(C1,C2,C3,C4,C5,C6->S1,S2,S3)`

There are only two test types:

--ping There are no private parameters for the ping test.

--brw The brw test can have several options:

read | write Read or write. The default is read.

size=# | #K | #M I/O size can be bytes, KB or MB (i.e., size=1024, size=4K, size=1M. The default is 4K bytes.

check=full | simple A data validation check (checksum of data). The default is no-check. As an example:

```
$ lst add_group clients 192.168.1.[10-17]@tcp
$ lst add_group servers 192.168.10.[100-103]@tcp
$ lst add_batch bulkperf
$ lst add_test --batch bulkperf --loop 100 \
--concurrency 4 --distribute 4:2 --from clients \
brw WRITE size=16K
// add brw (WRITE, 16 KB) test to batch bulkperf, \
the test will run in 4 workitem, each
// 192.168.1.[10-13] will write to
192.168.10.[100,101]
// 192.168.1.[14-17] will write to
192.168.10.[102,103]
```

list_batch [NAME] [--test INDEX] [--active] [--invalid] [--server]

Lists batches in the current session or lists client | server nodes in a batch or a test.

--test INDEX Lists tests in a batch. If no option is used, all tests in the batch are listed. If the option is used, only specified tests in the batch are listed.

```
$ lst list_batch
bulkperf
$ lst list_batch bulkperf
Batch: bulkperf Tests: 1 State: Idle
ACTIVE BUSY DOWN UNKNOWN TOTAL
client 8 0 0 0 8
server 4 0 0 0 4
Test 1(brw) (loop: 100, concurrency: 4)
ACTIVE BUSY DOWN UNKNOWN TOTAL
client 8 0 0 0 8
server 4 0 0 0 4
$ lst list_batch bulkperf --server --active
192.168.10.100@tcp Active
192.168.10.101@tcp Active
192.168.10.102@tcp Active
192.168.10.103@tcp Active
```

run NAME

Runs the batch.

```
$ lst run bulkperf
```

stop NAME

Stops the batch.

```
$ lst stop bulkperf
```

query NAME [--test INDEX] [--timeout #] [--loop #] [--delay #] [--all]

Queries the batch status.

--test INDEX	Only queries the specified test. The test INDEX starts from 1.
--timeout #	The timeout value to wait for RPC. The default is 5 seconds.
--loop #	The loop count of the query.
--delay #	The interval of each query. The default is 5 seconds.
--all	The list status of all nodes in a batch or a test.

```
$ lst run bulkperf
$ lst query bulkperf --loop 5 --delay 3
Batch is running
Batch is running
Batch is running
Batch is running
Batch is running
$ lst query bulkperf --all
192.168.1.10@tcp Running
192.168.1.11@tcp Running
192.168.1.12@tcp Running
192.168.1.13@tcp Running
192.168.1.14@tcp Running
192.168.1.15@tcp Running
192.168.1.16@tcp Running
192.168.1.17@tcp Running
$ lst stop bulkperf
$ lst query bulkperf
Batch is idle
```

18.4.2.4 Other Commands

This section lists other lst commands.

ping [-session] [--group NAME] [--nodes NIDs] [--batch name] [--server] [--timeout #]

Sends a “hello” query to the nodes.

–-session	Pings all nodes in the current session.
–-group NAME	Pings all nodes in a specified group.
–-nodes NIDs	Pings all specified nodes.
–-batch NAME	Pings all client nodes in a batch.
–-server	Sends RPC to all server nodes instead of client nodes. This option is only used with **batch NAME**.
–-timeout #	The RPC timeout value.

```
$ lst ping 192.168.10.[15-20]@tcp
192.168.1.15@tcp Active [session: liang id: 192.168.1.3@tcp]
192.168.1.16@tcp Active [session: liang id: 192.168.1.3@tcp]
192.168.1.17@tcp Active [session: liang id: 192.168.1.3@tcp]
192.168.1.18@tcp Busy [session: Isaac id: 192.168.10.10@tcp]
192.168.1.19@tcp Down [session: <NULL> id: LNET_NID_ANY]
192.168.1.20@tcp Down [session: <NULL> id: LNET_NID_ANY]
```

stat [--bw] [--rate] [--read] [--write] [--max] [--min] [--avg] " " [--timeout #] [--delay #] GROUP | NIDs [GROUP | NIDs]

The collection performance and RPC statistics of one or more nodes.

Specifying a group name (GROUP) causes statistics to be gathered for all nodes in a test group. For example:

```
$ lst stat servers
```

where `servers` is the name of a test group created by `lst add_group`

Specifying a NID range (NIDs) causes statistics to be gathered for selected nodes. For example:

```
$ lst stat 192.168.0.[1-100/2]@tcp
```

Currently, only LNET performance statistics are available.[2] By default, all statistics information is displayed. Users can specify additional information with these options.

--bw	Displays the bandwidth of the specified group/nodes.
--rate	Displays the rate of RPCs of the specified group/nodes.
--read	Displays the read statistics of the specified group/nodes.
--write	Displays the write statistics of the specified group/nodes.
--max	Displays the maximum value of the statistics.
--min	Displays the minimum value of the statistics.
--avg	Displays the average of the statistics.
--timeout #	The timeout of the statistics RPC. The default is 5 seconds.
--delay #	The interval of the statistics (in seconds).

```
$ lst run bulkperf
$ lst stat clients
[LNet Rates of clients]
[W] Avg: 1108 RPC/s Min: 1060 RPC/s Max: 1155 RPC/s
[R] Avg: 2215 RPC/s Min: 2121 RPC/s Max: 2310 RPC/s
[LNet Bandwidth of clients]
[W] Avg: 16.60 MB/s Min: 16.10 MB/s Max: 17.1 MB/s
[R] Avg: 40.49 MB/s Min: 40.30 MB/s Max: 40.68 MB/s
```

2. In the future, more statistics will be supported.

show_error [--session] [GROUP]|[NIDs] ...

Lists the number of failed RPCs on test nodes.

–-session Lists errors in the current test session. With this option, historical RPC errors are not listed.

```
$ lst show_error clients
clients
12345-192.168.1.15@tcp: [Session: 1 brw errors, 0 ping errors] \
[RPC: 20 errors, 0 dropped,
12345-192.168.1.16@tcp: [Session: 0 brw errors, 0 ping errors] \
[RPC: 1 errors, 0 dropped, Total 2 error nodes in clients
$ lst show_error --session clients
clients
12345-192.168.1.15@tcp: [Session: 1 brw errors, 0 ping errors]
Total 1 error nodes in clients
```

CHAPTER 19

Lustre Recovery

This chapter describes how to recover Lustre, and includes the following sections:

19.1 Recovery Overview

Lustre's recovery support is responsible to deal with node or network failure and return the cluster to a consistent, performant state. Because Lustre allows servers to perform asynchronous update operations to the on-disk file system (i.e. the server can reply without waiting for the update to synchronously commit to disk), the clients may have state in memory that is newer than what the server can recover from disk after a crash.

A handful of different types of failures can cause recovery to occur:

- Client (compute node) failure
- MDS failure (and failover)
- OST failure (and failover)
- Transient network partition

Currently, all Lustre failure and recovery operations are based on the concept of connection failure; all imports or exports associated with a given connection are considered to fail if any of them fail.

For information on Lustre recovery, see Metadata Replay. For information on recovering from a corrupt file system, see Recovering from Errors or Corruption on a Backing File System. For information on resolving orphaned objects, a common issue after recovery, see Working with Orphaned Objects.

19.1.1 Client Failure

Lustre's support for recovery from client failure is based on lock revocation and other resources, so surviving clients can continue their work uninterrupted. If a client fails to timely respond to a blocking AST from the Distributed Lock Manager (DLM) or fails to communicate with the server in a long period of time (i.e. no pings), the non-responding client is forcibly removed from the cluster. This enables other clients to acquire locks blocked by the non-responding client's locks, and also frees resources (file handles, export data) associated with the dead client. Note that this scenario can be caused by a network partition, as well as an actual client node system failure. Network Partition describes this case in more detail.

19.1.2 Client Eviction

If a client is not behaving properly from the server's point of view, it will be evicted. This ensures that the whole file system can continue to function in the presence of failed or misbehaving clients. An evicted client must invalidate all locks, which in turn, results in all cached inodes becoming invalidated and all cached data being flushed.

Reasons why a client might be evicted:

- Failure to respond to a server request in a timely manner
  - Blocking lock callback (i.e. client holds lock that another client/server wants)
  - Lock completion callback (i.e. client is granted lock previously held by another client)
  - Lock glimpse callback (i.e. client is asked for size of object by another client)
  - Server shutdown notification (with Simplified Interoperability)
- Failure to ping the server within a timely manner, unless the server is receiving no RPC traffic at all (which may indicate a network partition).

19.1.3 MDS Failure (Failover)

Highly-available Lustre operation requires that the metadata server have a peer configured for failover, including the use of a shared storage device for the MDT backing filesystem. It is also possible to have MDS recovery with a single MDS node. In this case, recovery will take as long as is needed for the single MDS to be restarted.

When clients detect an MDS failure, they connect to the new MDS and use the Metadata Replay protocol. Metadata Replay is responsible to ensure that the replacement MDS re-accumulates the state resulting from transactions whose effects were made visible to clients, but which were not committed to the disk.

The reconnection to a new (or restarted) MDS is managed by the file system configuration loaded by the client when the file system is first mounted. If a failover MDS has been configured (using the `--failnode=` option to `mkfs.lustre` or `tunefs.lustre`), the client tries to reconnect to both the primary and backup MDS until one of them responds that the failed MDT is again available. At that point, the client begins recovery. For more information, see Metadata Replay.

Transaction numbers are used to ensure that operations are replayed in the order they were originally performed, so that they are guaranteed to succeed and present the same filesystem state as before the failure. In addition, clients inform the new server of their existing lock state (including locks that have not yet been granted). All metadata and lock replay must complete before new, non-recovery operations are permitted. In addition, only clients that were connected at the time of MDS failure

are permitted to reconnect during the recovery window, to avoid the introduction of state changes that might conflict with what is being replayed by previously-connected clients.

19.1.4 OST Failure (Failover)

When an OST fails, or has communication problems with the client, the default action is that the corresponding OSC enters recovery, and IO requests going to that OST are blocked waiting for OST recovery or failover. It is possible to administratively mark the OSC as *inactive* on the client, in which case file operations that involve the failed OST will return an IO error (-EIO). Otherwise, the application will wait until the OST has recovered, or the client process is interrupted (e.g. with CTRL-C).

The MDS (via the LOV) detects that an OST is unavailable and skips it when assigning objects to new files. When the OST is restarted or re-establishes communication with the MDS, the MDS and OST automatically perform orphan recovery to destroy any objects that belong to files that were deleted while the OST was unavailable. For more information, see Working with Orphaned Objects.

While the OSC to OST operation recovery protocol is the same as that between the MDC and MDT using the Metadata Replay protocol, typically the OST commits bulk write operations to disk synchronously and each reply indicates that the request is already committed and the data does not need to be saved for recovery. In some cases, the OST replies to the client before the operation is committed to disk (e.g. truncate, destroy, setattr, and I/O operations in very new versions of Lustre), and normal replay and resend handling is done, including resending of the bulk writes. In this case, the client keeps a copy of the data available in memory until the server indicates that the write has committed to disk.

To force an OST recovery, unmount the OST and then mount it again. If the OST was connected to clients before it failed, then a recovery process starts after the remount, enabling clients to reconnect to the OST and replay transactions in their queue. When the OST is in recovery mode, all new client connections are refused until the recovery finishes. The recovery is complete when either all previously-connected clients reconnect and their transactions are replayed or a client connection attempt times out. If a connection attempt times out, then all clients waiting to reconnect (and their transactions) are lost.

Note – If you know an OST will not recover a previously-connected client (if, for example, the client has crashed), you can manually abort the recovery using this command:

```
lctl --device <OST device number> abort_recovery
```

To determine an OST's device number and device name, run the `lctl dl` command. Sample `lctl dl` command output is shown below:

```
7 UP obdfilter ddn_data-OST0009 ddn_data-OST0009_UUID 1159
```

In this example, `7` is the OST device number. The device name is `ddn_data-OST0009`. In most instances, the device name can be used in place of the device number.

19.1.5 Network Partition

Network failures may be transient. To avoid invoking recovery, the client tries, initially, to re-send any timed out request to the server. If the re-send also fails, the client tries to re-establish a connection to the server. Clients can detect harmless partition upon reconnect. If a reply was dropped, the client must reconstruct the reply. If a request was processed by the server, but the reply was dropped (i.e., did not arrive back at the client), the server must reconstruct the reply when the client resends the request, rather than performing the same request twice.

19.1.6 Failed Recovery

In the case of failed recovery, a client is evicted by the server and must reconnect after having flushed its saved state related to that server, as described in Client Eviction, above. Failed recovery might occur for a number of reasons, including:

- Failure of recovery
  - In Lustre 1.8, recovery fails if the operations of one client directly depend on the operations of another client that failed to participate in recovery. Otherwise, Version Based Recovery (VBR) allows recovery to proceed for all of the connected clients, and only missing clients are evicted.
  - Manual abort of recovery
- Manual eviction by the administrator

19.2 Metadata Replay

Highly available Lustre operation requires that the MDS have a peer configured for failover, including the use of a shared storage device for the MDS backing file system. When a client detects an MDS failure, it connects to the new MDS and uses the metadata replay protocol to replay its requests.

Metadata replay ensures that the failover MDS re-accumulates state resulting from transactions whose effects were made visible to clients, but which were not committed to the disk.

19.2.1 XID Numbers

Each request sent by the client contains an XID number, which is a client-unique, monotonically increasing 64-bit integer. The initial value of the XID is chosen so that it is highly unlikely that the same client node reconnecting to the same server after a reboot would have the same XID sequence. The XID is used by the client to order all of the requests that it sends, until such a time that the request is assigned a transaction number. The XID is also used in Reply Reconstruction to uniquely identify per-client requests at the server.

19.2.2 Transaction Numbers

Each client request processed by the server that involves any state change (metadata update, file open, write, etc., depending on server type) is assigned a transaction number by the server that is a target-unique, monontonically increasing, server-wide 64-bit integer. The transaction number for each file system-modifying request is sent back to the client along with the reply to that client request. The transaction numbers allow the client and server to unambiguously order every modification to the file system in case recovery is needed.

Each reply sent to a client (regardless of request type) also contains the last committed transaction number that indicates the highest transaction number committed to the file system. The backing file systems that Lustre uses (ext3/4, ZFS) enforce the requirement that any earlier disk operation will always be committed to disk before a later disk operation, so the last committed transaction number also reports that any requests with a lower transaction number have been committed to disk.

19.2.3 Replay and Resend

Lustre recovery can be separated into two distinct types of operations: *replay* and *resend*.

Replay operations are those for which the client received a reply from the server that the operation had been successfully completed. These operations need to be redone in exactly the same manner after a server restart as had been reported before the server failed. Replay can only happen if the server failed; otherwise it will not have lost any state in memory.

Resend operations are those for which the client never received a reply, so their final state is unknown to the client. The client sends unanswered requests to the server again in XID order, and again awaits a reply for each one. In some cases, resent requests have been handled and committed to disk by the server (possibly also having dependent operations committed), in which case, the server performs reply reconstruction for the lost reply. In other cases, the server did not receive the lost request at all and processing proceeds as with any normal request. These are what happen in the case of a network interruption. It is also possible that the server received the request, but was unable to reply or commit it to disk before failure.

19.2.4 Client Replay List

All file system-modifying requests have the potential to be required for server state recovery (replay) in case of a server failure. Replies that have an assigned transaction number that is higher than the last committed transaction number received in any reply from each server are preserved for later replay in a per-server replay list. As each reply is received from the server, it is checked to see if it has a higher last committed transaction number than the previous highest last committed number. Most requests that now have a lower transaction number can safely be removed from the replay list. One exception to this rule is for open requests, which need to be saved for replay until the file is closed so that the MDS can properly reference count open-unlinked files.

19.2.5 Server Recovery

A server enters recovery if it was not shut down cleanly. If, upon startup, if any client entries are in the last_rcvd file for any previously connected clients, the server enters recovery mode and waits for these previously-connected clients to reconnect and begin replaying or resending their requests. This allows the server to recreate state that was exposed to clients (a request that completed successfully) but was not committed to disk before failure.

In the absence of any client connection attempts, the server waits indefinitely for the clients to reconnect. This is intended to handle the case where the server has a network problem and clients are unable to reconnect and/or if the server needs to be restarted repeatedly to resolve some problem with hardware or software. Once the server detects client connection attempts - either new clients or previously-connected clients - a recovery timer starts and forces recovery to finish in a finite time regardless of whether the previously-connected clients are available or not.

If no client entries are present in the last_rcvd file, or if the administrator manually aborts recovery, the server does not wait for client reconnection and proceeds to allow all clients to connect.

As clients connect, the server gathers information from each one to determine how long the recovery needs to take. Each client reports its connection UUID, and the server does a lookup for this UUID in the last_rcvd file to determine if this client was previously connected. If not, the client is refused connection and it will retry until recovery is completed. Each client reports its last seen transaction, so the server knows when all transactions have been replayed. The client also reports the amount of time that it was previously waiting for request completion so that the server can estimate how long some clients might need to detect the server failure and reconnect.

If the client times out during replay, it attempts to reconnect. If the client is unable to reconnect, REPLAY fails and it returns to DISCON state. It is possible that clients will timeout frequently during REPLAY, so reconnection should not delay an already slow process more than necessary. We can mitigate this by increasing the timeout during replay.

19.2.6 Request Replay

If a client was previously connected, it gets a response from the server telling it that the server is in recovery and what the last committed transaction number on disk is. The client can then iterate through its replay list and use this last committed transaction number to prune any previously-committed requests. It replays any newer requests to the server in transaction number order, one at a time, waiting for a reply from the server before replaying the next request.

Open requests that are on the replay list may have a transaction number lower than the server's last committed transaction number. The server processes those open requests immediately. The server then processes replayed requests from all of the clients in transaction number order, starting at the last committed transaction number to ensure that the state is updated on disk in exactly the same manner as it was before the crash. As each replayed request is processed, the last committed transaction is incremented. If the server receives a replay request from a client that is higher than the current last committed transaction, that request is put aside until other clients provide the intervening transactions. In this manner, the server replays requests in the same sequence as they were previously executed on the server until either all clients are out of requests to replay or there is a gap in a sequence.

19.2.7 Gaps in the Replay Sequence

In some cases, a gap may occur in the reply sequence. This might be caused by lost replies, where the request was processed and committed to disk but the reply was not received by the client. It can also be caused by clients missing from recovery due to partial network failure or client death.

In the case where all clients have reconnected, but there is a gap in the replay sequence the only possibility is that some requests were processed by the server but the reply was lost. Since the client must still have these requests in its resend list, they are processed after recovery is finished.

In the case where all clients have not reconnected, it is likely that the failed clients had requests that will no longer be replayed. In Lustre 1.8 and later, version-based recovery (VBR) is used to determine if a request following a transaction gap is safe to be replayed. Each item in the file system (MDS inode or OST object) stores on disk the number of the last transaction in which it was modified. Each reply from the server contains the previous version number of the objects that it affects. During VBR replay, the server matches the previous version numbers in the resend request against the current version number. If the versions match, the request is the next one that affects the object and can be safely replayed. For more information, see Version-based Recovery.

19.2.8 Lock Recovery

If all requests were replayed successfully and all clients reconnected, clients then do lock replay locks -- that is, every client sends information about every lock it holds from this server and its state (whenever it was granted or not, what mode, what properties and so on), and then recovery completes successfully. Currently, Lustre does not do lock verification and just trusts clients to present an accurate lock state. This does not impart any security concerns since Lustre 1.x clients are trusted for other information (e.g. user ID) during normal operation also.

After all of the saved requests and locks have been replayed, the client sends an MDS_GETSTATUS request with last-replay flag set. The reply to that request is held back until all clients have completed replay (sent the same flagged getstatus request), so that clients don't send non-recovery requests before recovery is complete.

19.2.9 Request Resend

Once all of the previously-shared state has been recovered on the server (the target file system is up-to-date with client cache and the server has recreated locks representing the locks held by the client), the client can resend any requests that did not receive an earlier reply. This processing is done like normal request processing, and, in some cases, the server may do reply reconstruction.

19.3 Reply Reconstruction

When a reply is dropped, the MDS needs to be able to reconstruct the reply when the original request is re-sent. This must be done without repeating any non-idempotent operations, while preserving the integrity of the locking system. In the event of MDS failover, the information used to reconstruct the reply must be serialized on the disk in transactions that are joined or nested with those operating on the disk.

19.3.1 Required State

For the majority of requests, it is sufficient for the server to store three pieces of data in the `last_rcvd` file:

- XID of the request
- Resulting transno (if any)
- Result code (req->rq_status)

For open requests, the "disposition" of the open must also be stored.

19.3.2 Reconstruction of Open Replies

An open reply consists of up to three pieces of information (in addition to the contents of the "request log"):

- File handle
- Lock handle
- mds_body with information about the file created (for O_CREAT)

The disposition, status and request data (re-sent intact by the client) are sufficient to determine which type of lock handle was granted, whether an open file handle was created, and which resource should be described in the mds_body.

Finding the File Handle

The file handle can be found in the XID of the request and the list of per-export open file handles. The file handle contains the resource/FID.

Finding the Resource/fid

The file handle contains the resource/fid.

Finding the Lock Handle

The lock handle can be found by walking the list of granted locks for the resource looking for one with the appropriate remote file handle (present in the re-sent request). Verify that the lock has the right mode (determined by performing the disposition/request/status analysis above) and is granted to the proper client.

19.4 Version-based Recovery

Lustre 1.8 introduces the Version-based Recovery (VBR) feature, which improves Lustre reliability in cases where client requests (RPCs) fail to replay during recovery[1].

In pre-1.8 versions of Lustre, if the MGS or an OST went down and then recovered, a recovery process was triggered in which clients attempted to replay their requests. Clients were only allowed to replay RPCs in serial order. If a particular client could not replay its requests, then those requests were lost as well as the requests of clients later in the sequence. The "downstream" clients never got to replay their requests because of the wait on the earlier client's RPCs. Eventually, the recovery period would time out (so the component could accept new requests), leaving some number of clients evicted and their requests and data lost.

With VBR, the recovery mechanism does not result in the loss of clients or their data, because changes in inode versions are tracked, and more clients are able to reintegrate into the cluster. With VBR, inode tracking looks like this:

- Each inode[2] stores a version, that is, the number of the last transaction (transno) in which the inode was changed.
- When an inode is about to be changed, a pre-operation version of the inode is saved in the client's data.
- The client keeps the pre-operation inode version and the post-operation version (transaction number) for replay, and sends them in the event of a server failure.
- If the pre-operation version matches, then the request is replayed. The post-operation version is assigned on all inodes modified in the request.

Note – An RPC can contain up to four pre-operation versions, because several inodes can be involved in an operation. In the case of a "rename" operation, four different inodes can be modified.

---

1. There are two scenarios under which client RPCs are not replayed:

   (1) Non-functioning or isolated clients do not reconnect, and they cannot replay their RPCs, causing a gap in the replay sequence. These clients get errors and are evicted.

   (2) Functioning clients connect, but they cannot replay some or all of their RPCs that occurred after the gap caused by the non-functioning/isolated clients. These clients get errors (caused by the failed clients). With VBR, these requests have a better chance to replay because the "gaps" are only related to specific files that the missing client(s) changed.

2. Usually, there are two inodes, a parent and a child.

During normal operation, the server:

- Updates the versions of all inodes involved in a given operation
- Returns the old and new inode versions to the client with the reply

When the recovery mechanism is underway, VBR follows these steps:

1. **VBR only allows clients to replay transactions if the affected inodes have the same version as during the original execution of the transactions, even if there is gap in transactions due to a missed client.**
2. **The server attempts to execute every transaction that the client offers, even if it encounters a re-integration failure.**
3. **When the replay is complete, the client and server check if a replay failed on any transaction because of inode version mismatch. If the versions match, the client gets a successful re-integration message. If the versions do not match, then the client is evicted.**

VBR recovery is fully transparent to users. It may lead to slightly longer recovery times if the cluster loses several clients during server recovery.

19.4.1 Delayed Recovery

With VBR, it is possible to recover clients even after the server's recovery window closes. This is known as delayed recovery. This feature is useful if clients have become temporarily unavailable during recovery (e.g., because of a network partition).

Note – In Lustre 1.8, the delayed recovery feature is available as a preview, and is turned off by default. It is designed for use with future versions of Lustre, to help with disconnected operations.

19.4.2 Working with VBR

In Lustre 1.8, the VBR feature is built into the Lustre recovery functionality. It cannot be disabled.

Delayed recovery can be enabled with the `--enable-delayed-recovery` option:

```
./configure ... --enable-delayed-recovery
```

During reboot, a list of new messages is displayed.

```
CWARN("RECOVERY: service %s, %d recoverable clients, last_transno
"LPU64"\n"); was updated with number delayed clients:

CWARN("RECOVERY: service %s, %d recoverable clients, %d delayed
clients, last_transno "LPU64"\n");
```

Note – There should be no delayed clients until delayed recovery is enabled.

These are some VBR messages that may be displayed:

```
DEBUG_REQ(D_WARNING, req, "Version mismatch during replay\n");
```

This message indicates why the client was evicted. No action is needed.

```
CWARN("%s: version recovery fails, reconnecting\n");
```

This message indicates why the recovery failed. No action is needed.

These are some VBR messages that may be displayed if delayed recovery is enabled:

```
CWARN("RECOVERY: service %s, %d recoverable clients, %d delayed
clients, last_transno "LPU64"\n");
```

This controls the number of delayed clients. There should be 0 delayed clients without delayed recovery enabled).

```
CWARN("%s: NID %s (%s) export was already marked as delayed and will
wait for end of recovery\n");
```

The old client is trying to reconnect, but it will wait for end of the server's recovery period. No action is needed.

19.4.3 Tips for Using VBR

VBR will be successful for clients which do not share data with other client. Therefore, the strategy for reliable use of VBR is to store a client's data in its own directory, where possible. VBR can recover these clients, even if other clients are lost.

19.5 Recovering from Errors or Corruption on a Backing File System

When an OSS, MDS, or MGS server crash occurs, it is not necessary to run e2fsck on the file system. Ext3 journaling ensures that the file system remains coherent. The backing file systems are never accessed directly from the client, so client crashes are not relevant.

The only time it is REQUIRED that e2fsck be run on a device is when an event causes problems that ext3 journaling is unable to handle, such as a hardware device failure or I/O error. If the ext3 kernel code detects corruption on the disk, it mounts the file system as read-only to prevent further corruption, but still allows read access to the device. This appears as error "-30" (EROFS) in the syslogs on the server, e.g.:

```
Dec 29 14:11:32 mookie kernel: LDISKFS-fs error (device sdz):
ldiskfs_lookup: unlinked inode 5384166 in dir #145170469
```

Dec 29 14:11:32 mookie kernel: Remounting filesystem read-only

In such a situation, it is normally required that e2fsck only be run on the bad device before placing the device back into service.

In the vast majority of cases, Lustre can cope with any inconsistencies it finds on the disk and between other devices in the file system.

Note – `lfsck` is rarely required for Lustre operation.

For problem analysis, it is strongly recommended that e2fsck be run under a logger, like script, to record all of the output and changes that are made to the file system in case this information is needed later.

If time permits, it is also a good idea to first run e2fsck in non-fixing mode (`-n` option) to assess the type and extent of damage to the file system. The drawback is that in this mode, `e2fsck` does not recover the file system journal, so there may appear to be file system corruption when none really exists.

To address concern about whether corruption is real or only due to the journal not being replayed, you can briefly mount and unmount the ext3 filesystem directly on the node with Lustre stopped (NOT via Lustre), using a command similar to:

```
mount -t ldiskfs /dev/{ostdev} /mnt/ost; umount /mnt/ost
```

This causes the journal to be recovered.

The e2fsck utility works well when fixing file system corruption (better than similar file system recovery tools and a primary reason why ext3 was chosen over other file systems for Lustre). However, it is often useful to identify the type of damage that has occurred so an ext3 expert can make intelligent decisions about what needs fixing, in place of e2fsck.

```
root# {stop lustre services for this device, if running}
root# script /tmp/e2fsck.sda
Script started, file is /tmp/e2fsck.sda
root# mount -t ldiskfs /dev/sda /mnt/ost
root# umount /mnt/ost
root# e2fsck -fn /dev/sda   # don't fix file system, just check for
corruption
:
[e2fsck output]
:
root# e2fsck -fp /dev/sda   # fix filesystem using "prudent" answers
(usually 'y')
```

In addition, the e2fsprogs package contains the lfsck tool, which does distributed coherency checking for the Lustre file system after e2fsck has been run. Running lfsck is NOT required in a large majority of cases, at a small risk of having some leaked space in the file system. To avoid a lengthy downtime, it can be run (with care) after Lustre is started.

19.6 Recovering from Corruption in the Lustre File System

In cases where the MDS or an OST becomes corrupt, you can run a distributed check on the file system to determine what sort of problems exist. Use `lfsck` to correct any defects found.

1. **Stop the Lustre file system.**
2. **Run `e2fsck -f` on the individual MDS / OST that had problems to fix any local file system damage.**

   We recommend running e2fsck under script, to create a log of changes made to the file system in case it is needed later. After e2fsck is run, bring up the file system, if necessary, to reduce the outage window.
3. **Run a full e2fsck of the MDS to create a database for lfsck. It is critical to use the `-n` option for a mounted file system, otherwise you will corrupt the file system.**

   ```
   e2fsck -n -v --mdsdb /tmp/mdsdb /dev/{mdsdev}
   ```

   The mdsdb file can grow fairly large, depending on the number of files in the file system (10 GB or more for millions of files, though the actual file size is larger because the file is sparse). It is quicker to write the file to a local file system due to seeking and small writes. Depending on the number of files, this step can take several hours to complete.

   Example

   ```
   e2fsck -n -v --mdsdb /tmp/mdsdb /dev/sdb
   e2fsck 1.39.cfs1 (29-May-2006)
   Warning: skipping journal recovery because doing a read-only
   filesystem check.
   lustre-MDT0000 contains a file system with errors, check forced.
   Pass 1: Checking inodes, blocks, and sizes
   MDS: ost_idx 0 max_id 288
   MDS: got 8 bytes = 1 entries in lov_objids
   MDS: max_files = 13
   MDS: num_osts = 1
   mds info db file written
   Pass 2: Checking directory structure
   Pass 3: Checking directory connectivity
   Pass 4: Checking reference counts
   Pass 5: Checking group summary information
   Free blocks count wrong (656160, counted=656058).
   ```

```
Fix? no

Free inodes count wrong (786419, counted=786036).
Fix? no

Pass 6: Acquiring information for lfsck
MDS: max_files = 13
MDS: num_osts = 1
MDS: 'lustre-MDT0000_UUID' mdt idx 0: compat 0x4 rocomp 0x1 incomp
0x4
lustre-MDT0000: ******* WARNING: Filesystem still has errors
*******
           13 inodes used (0%)
               2 non-contiguous inodes (15.4%)
                   # of inodes with ind/dind/tind blocks: 0/0/0
130272 blocks used (16%)
           0 bad blocks
           1 large file
           296 regular files
           91 directories
           0 character device files
           0 block device files
           0 fifos
           0 links
           0 symbolic links (0 fast symbolic links)
           0 sockets
           --------
           387 files
```

4. **Make this file accessible on all OSTs, either by using a shared file system or copying the file to the OSTs. The `pdcp` command is useful here.**

   The `pdcp` command (installed with pdsh), can be used to copy files to groups of hosts. `Pdcp` is available here:

   http://sourceforge.net/projects/pdsh

5. **Run a similar e2fsck step on the OSTs. The `e2fsck --ostdb` command can be run in parallel on all OSTs.**

   ```
   e2fsck -n -v --mdsdb /tmp/mdsdb --ostdb /tmp/{ostNdb} \
   /dev/{ostNdev}
   ```

   The mdsdb file is read-only in this step; a single copy can be shared by all OSTs.

Note – If the OSTs do not have shared file system access to the MDS, a stub mdsdb file, {mdsdb}.mdshdr, is generated. This can be used instead of the full mdsdb file.

Example:

```
[root@oss161 ~]# e2fsck -n -v --mdsdb /tmp/mdsdb --ostdb \
/tmp/ostdb /dev/sda
e2fsck 1.39.cfs1 (29-May-2006)
Warning: skipping journal recovery because doing a read-only
filesystem check.
lustre-OST0000 contains a file system with errors, check forced.
Pass 1: Checking inodes, blocks, and sizes
Pass 2: Checking directory structure
Pass 3: Checking directory connectivity
Pass 4: Checking reference counts
Pass 5: Checking group summary information
Free blocks count wrong (989015, counted=817968).
Fix? no

Free inodes count wrong (262088, counted=261767).
Fix? no

Pass 6: Acquiring information for lfsck
OST: 'lustre-OST0000_UUID' ost idx 0: compat 0x2 rocomp 0 incomp 0x2
OST: num files = 321
OST: last_id = 321

lustre-OST0000: ******* WARNING: Filesystem still has errors
*******

           56 inodes used (0%)
           27 non-contiguous inodes (48.2%)
              # of inodes with ind/dind/tind blocks: 13/0/0
59561 blocks used (5%)
           0 bad blocks
           1 large file
           329 regular files
           39 directories
           0 character device files
           0 block device files
           0 fifos
           0 links
           0 symbolic links (0 fast symbolic links)
           0 sockets
           --------
           368 files
```

6. **Make the mdsdb file and all ostdb files available on a mounted client and run `lfsck` to examine the file system. Optionally, correct the defects found by lfsck.**

```
script /root/lfsck.lustre.log
lfsck -n -v --mdsdb /tmp/mdsdb --ostdb /tmp/{ost1db} /tmp/{ost2db}
... /lustre/mount/point
```

Example:

```
script /root/lfsck.lustre.log
lfsck -n -v --mdsdb /home/mdsdb --ostdb /home/{ost1db} \
/mnt/lustre/client/
MDSDB: /home/mdsdb
OSTDB[0]: /home/ostdb
MOUNTPOINT: /mnt/lustre/client/
MDS: max_id 288 OST: max_id 321
lfsck: ost_idx 0: pass1: check for duplicate objects
lfsck: ost_idx 0: pass1 OK (287 files total)
lfsck: ost_idx 0: pass2: check for missing inode objects
lfsck: ost_idx 0: pass2 OK (287 objects)
lfsck: ost_idx 0: pass3: check for orphan objects
[0] uuid lustre-OST0000_UUID
[0] last_id 288
[0] zero-length orphan objid 1
lfsck: ost_idx 0: pass3 OK (321 files total)
lfsck: pass4: check for duplicate object references
lfsck: pass4 OK (no duplicates)
lfsck: fixed 0 errors
```

By default, `lfsck` reports errors, but it does not repair any inconsistencies found. lfsck checks for three kinds of inconsistencies:

- Inode exists but has missing objects (dangling inode). This normally happens if there was a problem with an OST.
- Inode is missing but OST has unreferenced objects (orphan object). Normally, this happens if there was a problem with the MDS.
- Multiple inodes reference the same objects. This can happen if the MDS is corrupted or if the MDS storage is cached and loses some, but not all, writes.

If the file system is in use and being modified while the `--mdsdb` and `--ostdb` steps are running, `lfsck` may report inconsistencies where none exist due to files and objects being created/removed after the database files were collected. Examine the lfsck results closely. You may want to re-run the test.

19.6.1 Working with Orphaned Objects

The easiest problem to resolve is that of orphaned objects. When the `-l` option for `lfsck` is used, these objects are linked to new files and put into lost+found in the Lustre file system, where they can be examined and saved or deleted as necessary. If you are certain the objects are not useful, run `lfsck` with the `-d` option to delete orphaned objects and free up any space they are using.

To fix dangling inodes, use `lfsck` with the `-c` option to create new, zero-length objects on the OSTs. These files read back with binary zeros for stripes that had objects re-created. Even without `lfsck` repair, these files can be read by entering:

```
dd if=/lustre/bad/file of=/new/file bs=4k conv=sync,noerror
```

Because it is rarely useful to have files with large holes in them, most users delete these files after reading them (if useful) and/or restoring them from backup.

Note – You cannot write to the holes of such files without having `lfsck` re-create the objects. Generally, it is easier to delete these files and restore them from backup.

To fix inodes with duplicate objects, use `lfsck` with the `-c` option to copy the duplicate object to a new object and assign it to a file. One file will be okay and the duplicate will likely contain garbage. By itself, `lfsck` cannot tell which file is the usable one.

PART III Lustre Tuning, Monitoring and Troubleshooting

The part includes chapters describing how to tune, debug and troubleshoot Lustre.

CHAPTER 20

Lustre Tuning

This chapter contains information to tune Lustre for better performance and includes the following sections:

20.1 Module Options

Many options in Lustre are set by means of kernel module parameters. These parameters are contained in the `modprobe.conf` file (On SuSE, this may be `modprobe.conf.local`).

20.1.1 OSS Service Thread Count

The `oss_num_threads` parameter enables the number of OST service threads to be specified at module load time on the OSS nodes:

```
options ost oss_num_threads={N}
```

After startup, the minimum and maximum number of OSS thread counts can be set via the `{service}.thread_{min,max,started}` tunable. To change the tunable at runtime, run:

```
lctl {get,set}_param {service}.thread_{min,max,started}
```

For details, see Setting MDS and OSS Thread Counts.

20.1.1.1 Optimizing the Number of Service Threads

An OSS can have a minimum of 2 service threads and a maximum of 512 service threads. The number of service threads is a function of how much RAM and how many CPUs are on each OSS node (1 thread / 128MB * num_cpus). If the load on the OSS node is high, new service threads will be started in order to process more requests concurrently, up to 4x the initial number of threads (subject to the maximum of 512). For a 2GB 2-CPU system, the default thread count is 32 and the maximum thread count is 128.

Increasing the size of the thread pool may help when:

- Several OSTs are exported from a single OSS
- Back-end storage is running synchronously
- I/O completions take excessive time due to slow storage

Decreasing the size of the thread pool may help if:

- Clients are overwhelming the storage capacity
- There are lots of "slow I/O" or similar messages

Increasing the number of I/O threads allows the kernel and storage to aggregate many writes together for more efficient disk I/O. The OSS thread pool is shared—each thread allocates approximately 1.5 MB (maximum RPC size + 0.5 MB) for internal I/O buffers.

It is very important to consider memory consumption when increasing the thread pool size. Drives are only able to sustain a certain amount of parallel I/O activity before performance is degraded, due to the high number of seeks and the OST threads just waiting for I/O. In this situation, it may be advisable to decrease the load by decreasing the number of OST threads.

The optimum number of OST threads varies for each particular configuration. Variables include the number of OSTs on each OSS, number and speed of disks, RAID configuration, and available RAM. You may want to start with a number of OST threads equal to twice the number of actual disks on the node. If you use RAID5 or RAID6, subtract any disks used for parity (1 for RAID5, 2 for RAID6). Monitor the aggregate OSS I/O performance and increase the number of OST service threads (with `lctl set_param ost.OSS.ost_io.threads_max=N`) until throughput is maximized. Note that if there are too many threads, the latency for individual I/O requests can become very high and should be avoided. Set the desired maximum thread count permanently using the method described above.

20.1.2 MDS Service Thread Count

The `mds_num_threads` parameter enables the number of MDS service threads to be specified at module load time on the MDS node:

```
options mds mds_num_threads={N}
```

After startup, the minimum and maximum number of MDS thread counts can be set via the `{service}.thread_{min,max,started}` tunable. To change the tunable at runtime, run:

```
lctl {get,set}_param {service}.thread_{min,max,started}
```

For details, see Setting MDS and OSS Thread Counts.

At this time, we have not tested to determine the optimal number of MDS threads are. The default value varies, based on server size, up to a maximum of 32. The maximum number of threads (MDS_MAX_THREADS) is 512.

Note – The OSS and MDS automatically start new service threads dynamically, in response to server load, within a factor of 4. The default is calculated the same way as before (as explained in OSS Service Thread Count). Setting the `_mu_threads module` parameter disables automatic thread creation behavior.

20.1.2.1 I/O Scheduler

Benchmark all I/O schedulers and select the one that works best for your Lustre environment.

The Deadline scheduler is most likely to work well for Lustre OSTs, but NOOP could be considered for OSTs that are CPU-limited and use hardware RAID devices with lots of cache. For some workloads, the CFQ scheduler may also improve performance.

For more information on I/O schedulers, see:

http://www.linuxjournal.com/article/6931

http://www.redhat.com/magazine/008jun05/features/schedulers/

20.2 LNET Tunables

This section describes LNET tunables.

20.2.0.1 Transmit and receive buffer size:

With Lustre release 1.4.7 and later, ksocklnd now has separate parameters for the transmit and receive buffers.

```
options ksocklnd tx_buffer_size=0 rx_buffer_size=0
```

If these parameters are left at the default value (0), the system automatically tunes the transmit and receive buffer size. In almost every case, this default produces the best performance. Do not attempt to tune these parameters unless you are a network expert.

20.2.0.2 irq_affinity

By default, this parameter is on. In the normal case on an SMP system, we would like network traffic to remain local to a single CPU. This helps to keep the processor cache warm and minimizes the impact of context switches. This is especially helpful when an SMP system has more than one network interface and ideal when the number of interfaces equals the number of CPUs.

If you have an SMP platform with a single fast interface such as 10GB Ethernet and more than two CPUs, you may see performance improve by turning this parameter off, as always test to compare the impact.

20.3 Options for Formatting the MDT and OSTs

The backing file systems on an MDT and OSTs are independent of one another, so the formatting parameters for them should not be same. The size of the MDS backing file system depends solely on how many inodes you want in the total Lustre file system. It is not related to the size of the aggregate OST space.

20.3.1 Planning for Inodes

Each time you create a file on a Lustre file system, it consumes one inode on the MDT and one inode for each OST object that the file is striped over. Normally, it is based on the stripe count used for files, either from the file system-wide default set with --param lov.stripecount=N at mkfs.lustre time, or from the per-file or per-directory stripe count set with lfs --count=N. In ext3/ldiskfs file systems, inodes are pre-allocated, so creating a new file does not consume any of the free blocks. However, this also means that the format-time options should be conservative, as it is not possible to increase the number of inodes after the file system is formatted. If there is a shortage of inodes or space on the OSTs, it is possible to add OSTs to the file system.

To be on the safe side, plan for 4 KB per inode on the MDT (the default). For the OST, the amount of space taken by each object depends entirely upon the usage pattern of the users/applications running on the system. Lustre, by necessity, defaults to a very conservative estimate for the object size (16 KB per object). You can almost always increase this value for file system installations. Many Lustre file systems have average file sizes over 1 MB per object.

20.3.2 Sizing the MDT

When calculating the MDT size, the only important factor is the average size of files to be stored in the file system. If the average file size is, for example, 5 MB and you have 100 TB of usable OST space, then you need at least (100 TB * 1024 GB/TB * 1024 MB/GB / 5 MB/inode) = 20 million inodes. We recommend that you have twice the minimum (40 million inodes in this example). At the default 4 KB per inode, this works out to only 160 GB of space for the MDT.

Conversely, if you have a very small average file size (4 KB for example), Lustre is not very efficient. This is because you consume as much space on the MDT as on the OSTs. This is not a very common configuration for Lustre.

20.4 Overriding Default Formatting Options

To override the default formatting options for any of the Lustre backing filesystems, use the `--mkfsoptions='backing fs options'` argument to `mkfs.lustre` to pass formatting options to the backing mkfs. For all options to format backing ext3 and ldiskfs filesystems, see the `mke2fs(8)` man page; this section only discusses some Lustre-specific options.

20.4.1 Number of Inodes for the MDT

The number of inodes on the MDT is determined at format time based on the total size of the filesystem to be created. The default MDT inode ratio is one inode for every 4096 bytes of filesystem space. To override the inode ratio, use the option -i <bytes per inode>. For example, use --mkfsoptions="-i 2048" to create one inode per 2048 bytes of file system space. You should not specify the -i option with an inode ratio below one inode per 1024 bytes in order to avoid running out of space in the MDT filesystem for directories and Lustre internal metadata.

Alternately, to specify an total number of inodes, use the -N<number of inodes> option. It is still important to avoid specifying a total number of inodes that would consume too much space in the MDT filesystem with inode tables and not leave enough space for other filesystem metadata.

For example, a 2 TB MDS by default will have 512M inodes. The largest currently-supported file system size is 16 TB, which would hold 4B inodes, the maximum possible number of inodes with ldiskfs. Using an MDS inode ratio of 1024 bytes per inode, a 2 TB MDS would hold 2B inodes, and a 4 TB MDS would hold 4B inodes.

Reducing the inode ratio will also reduce the number of drives needed to hold a given number of inodes. For spinning disks this may be a disadvantage, since the number of IO operations per second (IOPS) will decrease proportionately and hurt MDS performance. For solid-state disks (SSDs) this can be advantageous, since the cost of SSDs is still quite high, and the number of IOPS per SSD is already very high.

20.4.2 Inode Size for the MDT

Lustre uses "large" inodes on backing file systems to efficiently store Lustre metadata with each file. On the MDT, each inode is at least 512 bytes in size (by default), while on the OST each inode is 256 bytes in size.

The inode size is specified at format time. Specifying a larger inode size may be useful in case the default Lustre stripe count is expected to increase in the future, or in case widespread use of Access Control Lists (ACLs) or user Extended Attributes (EAs) is anticipated. To increase the inode size, add the "-I<inodesize>" option to --mkfsoptions at format time.

We do NOT recommend specifying a smaller-than-default inode size, as this can lead to serious performance problems. If the inode size is too small, the MDS needs to make an indirect allocation to hold the Lustre striping EAs, and others, which permanently impacts performance due to additional seeks for accessing each file. The inode ratio must always be larger than the inode size as specified by the "-i" option previously discussed.

20.4.3 Number of Inodes for an OST

For OST file systems, it is normally advantageous to take local file system usage into account. Try to minimize the number of inodes on each OST. This helps reduce the format and `e2fsck` time, and makes more space available for data.

Note – In addition to the number of inodes, `e2fsck` runtime on OSTs is affected by a number of other variables: size of the file system, number of allocated blocks, distribution of allocated blocks on the disk, disk speed, CPU speed, and amount of RAM on the server. Reasonable `e2fsck` runtimes (without serious file system problems), are expected to take five minutes to two hours.

Presently, Lustre has 1 inode per 16 KB of space in the OST file system (by default). In many environments, this is too many inodes for the average file size. As a general guideline, OSTs should have at least a number of inodes indicated by this formula:

```
num_ost_inodes = 4 * <num_mds_inodes> * <default_stripe_count> <number_osts>
```

To specify the number of inodes on OST file systems, use the `-N<num_inodes>` option to `--mkfsoptions`. Alternately, if you know the average file size, you can also specify the OST inode count for the OST file systems using: `-i <average_file_size / (number_of_stripes * 4)>`. For example, if the average file size is 16 MB and there are, by default, 4 stripes per file then `--mkfsoptions=-i 1048576` would be appropriate.) .

For more details on formatting MDT and OST file systems, see Formatting Options for RAID Devices.

20.5 Large-Scale Tuning for Cray XT and Equivalents

This section only applies to Cray XT3 Catamount nodes, and explains parameters used with the kptllnd module. If it does not apply to your setup, ignore it.

20.5.1 Network Tunables

With a large number of clients and servers possible on these systems, tuning various request pools becomes important. We are making changes to the ptllnd module.

Parameter	Description
max_nodes	max_nodes is the maximum number of queue pairs, and, therefore, the maximum number of peers with which the LND instance can communicate. Set max_nodes to a value higher than the product of the total number of nodes and maximum processes per node. **Max nodes > (Total # Nodes) * (max_procs_per_node)** Setting max_nodes to a lower value than described causes Lustre to throw an error. Setting max_nodes to a higher value, causes excess memory to be consumed.
max_procs_per_node	max_procs_per_node is the maximum number of cores (CPUs), on a single Catamount node. Portals must know this value to properly clean up various queues. LNET is not notified directly when a Catamount process aborts. The first information LNET receives is when a new Catamount process with the same Cray portals NID starts and sends a connection request. If the number of processes with that Cray portals NID exceeds the `max_procs_per_node` value, LNET removes the oldest one to make space for the new one.

Parameter	Description
These two tunables combine to set the size of the ptllnd request buffer pool. The buffer pool must never drop an incoming message, so proper sizing is very important.	
Ntx	Ntx helps to size the transmit (tx) descriptor pool. A tx descriptor is used for each send and each passive RDMA. The max number of concurrent sends == 'credits'. Passive RDMA is a response to a PUT or GET of a payload that is too big to fit in a small message buffer. For servers, this only happens on large RPCs (for instance, where a long file name is included), so the MDS could be under pressure in a large cluster. For routers, this is bounded by the number of servers. If the tx pool is exhausted, a console error message appears.
Credits	Credits determine how many sends are in-flight at once on ptllnd. Optimally, there are 8 requests in-flight per server. The default value is 128, which should be adequate for most applications.

20.6 Lockless I/O Tunables

The lockless I/O tunable feature allows servers to ask clients to do lockless I/O (liblustre-style where the server does the locking) on contended files.

The lockless I/O patch introduces these tunables:

- OST-side:

  `/proc/fs/lustre/ldlm/namespaces/filter-lustre-*`

  `contended_locks` - If the number of lock conflicts in the scan of granted and waiting queues at `contended_locks` is exceeded, the resource is considered to be contended.

  `contention_seconds` - The resource keeps itself in a contended state as set in the parameter.

  `max_nolock_bytes` - Server-side locking set only for requests less than the blocks set in the `max_nolock_bytes` parameter. If this tunable is set to zero (0), it disables server-side locking for read/write requests.

- Client-side:

  `/proc/fs/lustre/llite/lustre-*`

  `contention_seconds` - llite inode remembers its contended state for the time specified in this parameter.

- Client-side statistics:

  The `/proc/fs/lustre/llite/lustre-*/stats` file has new rows for lockless I/O statistics.

  `lockless_read_bytes` and `lockless_write_bytes` - To count the total bytes read or written, the client makes its own decisions based on the request size. The client does not communicate with the server if the request size is smaller than the `min_nolock_size`, without acquiring locks by the client.

20.7 Data Checksums

To avoid the risk of data corruption on the network, a Lustre client can perform end-to-end data checksums[1]. Be aware that at high data rates, checksumming can impact Lustre performance.

1. This feature computes a 32-bit checksum of data read or written on both the client and server, and ensures that the data has not been corrupted in transit over the network.

CHAPTER 21

LustreProc

This chapter describes Lustre `/proc` entries and includes the following sections:

The proc file system acts as an interface to internal data structures in the kernel. Proc variables can be used to control aspects of Lustre performance and provide information.

21.1 Proc Entries for Lustre

This section describes /proc entries for Lustre.

21.1.1 Locating Lustre File Systems and Servers

Use the proc files on the MGS to locate the following:

- All known file systems

```
# cat /proc/fs/lustre/mgs/MGS/filesystems
spfs
lustre
```

- The server names participating in a file system (for each file system that has at least one server running)

```
# cat /proc/fs/lustre/mgs/MGS/live/spfs
fsname: spfs
flags: 0x0     gen: 7
spfs-MDT0000
spfs-OST0000
```

All servers are named according to this convention: *<fsname>-<MDT|OST><XXXX>* This can be shown for live servers under /proc/fs/lustre/devices:

```
# cat /proc/fs/lustre/devices
0 UP mgs MGS MGS 11
1 UP mgc MGC192.168.10.34@tcp 1f45bb57-d9be-2ddb-c0b0-5431a49226705
2 UP mdt MDS MDS_uuid 3
3 UP lov lustre-mdtlov lustre-mdtlov_UUID 4
4 UP mds lustre-MDT0000 lustre-MDT0000_UUID 7
5 UP osc lustre-OST0000-osc lustre-mdtlov_UUID 5
6 UP osc lustre-OST0001-osc lustre-mdtlov_UUID 5
7 UP lov lustre-clilov-ce63ca00 08ac6584-6c4a-3536-2c6d-b36cf9cbdaa04
8 UP mdc lustre-MDT0000-mdc-ce63ca00
08ac6584-6c4a-3536-2c6d-b36cf9cbdaa05
9 UP osc lustre-OST0000-osc-ce63ca00
08ac6584-6c4a-3536-2c6d-b36cf9cbdaa05
10 UP osc lustre-OST0001-osc-ce63ca00
08ac6584-6c4a-3536-2c6d-b36cf9cbdaa05
```

Or from the device label at any time:

```
# e2label /dev/sda
lustre-MDT0000
```

21.1.2 Lustre Timeouts

Lustre uses two types of timeouts.

- LND timeouts that ensure point-to-point communications complete in finite time in the presence of failures. These timeouts are logged with the S_LND flag set. They may not be printed as console messages, so you should check the Lustre log for D_NETERROR messages, or enable printing of D_NETERROR messages to the console (echo + neterror > /proc/sys/lnet/printk).

  Congested routers can be a source of spurious LND timeouts. To avoid this, increase the number of LNET router buffers to reduce back-pressure and/or increase LND timeouts on all nodes on all connected networks. You should also consider increasing the total number of LNET router nodes in the system so that the aggregate router bandwidth matches the aggregate server bandwidth.
- Lustre timeouts that ensure Lustre RPCs complete in finite time in the presence of failures. These timeouts should always be printed as console messages. If Lustre timeouts are not accompanied by LNET timeouts, then you need to increase the lustre timeout on both servers and clients.

Specific Lustre timeouts are described below.

/proc/sys/lustre/timeout

This is the time period that a client waits for a server to complete an RPC (default is 100s). Servers wait half of this time for a normal client RPC to complete and a quarter of this time for a single bulk request (read or write of up to 1 MB) to complete. The client pings recoverable targets (MDS and OSTs) at one quarter of the timeout, and the server waits one and a half times the timeout before evicting a client for being "stale."

Note – Lustre sends periodic 'PING' messages to servers with which it had no communication for a specified period of time. Any network activity on the file system that triggers network traffic toward servers also works as a health check.

/proc/sys/lustre/ldlm_timeout

This is the time period for which a server will wait for a client to reply to an initial AST (lock cancellation request) where default is 20s for an OST and 6s for an MDS. If the client replies to the AST, the server will give it a normal timeout (half of the client timeout) to flush any dirty data and release the lock.

/proc/sys/lustre/fail_loc

This is the internal debugging failure hook.

See `lustre/include/linux/obd_support.h` for the definitions of individual failure locations. The default value is 0 (zero).

```
sysctl -w lustre.fail_loc=0x80000122 # drop a single reply
```

/proc/sys/lustre/dump_on_timeout

This triggers dumps of the Lustre debug log when timeouts occur. The default value is 0 (zero).

/proc/sys/lustre/dump_on_eviction

This triggers dumps of the Lustre debug log when an eviction occurs. The default value is 0 (zero). By default, debug logs are dumped to the /tmp folder; this location can be changed via /proc.

21.1.3 Adaptive Timeouts

Lustre 1.8 introduces an adaptive mechanism to set RPC timeouts. This feature causes servers to track actual RPC completion times, and to report estimated completion times for future RPCs back to clients. The clients use these estimates to set their future RPC timeout values. If server request processing slows down for any reason, the RPC completion estimates increase, and the clients allow more time for RPC completion.

If RPCs queued on the server approach their timeouts, then the server sends an early reply to the client, telling the client to allow more time. In this manner, clients avoid RPC timeouts and disconnect/reconnect cycles. Conversely, as a server speeds up, RPC timeout values decrease, allowing faster detection of non-responsive servers and faster attempts to reconnect to a server's failover partner.

Note – In Lustre 1.8, adaptive timeouts are enabled, by default. In earlier Lustre versions supporting adaptive timeouts (1.6.5 through 1.6.7.x), this feature was disabled, by default.

In previous Lustre versions, the static `obd_timeout` (/proc/sys/lustre/timeout) value was used as the maximum completion time for all RPCs; this value also affected the client-server ping interval and initial recovery timer. Now, with adaptive timeouts, `obd_timeout` is only used for the ping interval and initial recovery estimate. When a client reconnects during recovery, the server uses the client's timeout value to reset the recovery wait period; i.e., the server learns how long the client had been willing to wait, and takes this into account when adjusting the recovery period.

21.1.3.1 Configuring Adaptive Timeouts

One of the goals of adaptive timeouts is to relieve users from having to tune the `obd_timeout` value. In general, `obd_timeout` should no longer need to be changed. However, there are several parameters related to adaptive timeouts that users can set. In most situations, the default values should be used.

The following parameters can be set persistently system-wide using `lctl conf_param` on the MGS. For example, `lctl conf_param work1.sys.at_max= 1500` sets the `at_max` value for all servers and clients using the `work1` file system.

Note – Nodes using multiple Lustre file systems must use the same at_* values for all file systems.)

Parameter	Description
at_min	Sets the minimum adaptive timeout (in seconds). Default value is 0. The `at_min` parameter is the minimum processing time that a server will report. Clients base their timeouts on this value, but they do not use this value directly. If you experience cases in which, for unknown reasons, the adaptive timeout value is too short and clients time out their RPCs (usually due to temporary network outages), then you can increase the `at_min` value to compensate for this. Ideally, users should leave `at_min` set to its default.
at_max	Sets the maximum adaptive timeout (in seconds). The `at_max` parameter is an upper-limit on the service time estimate, and is used as a 'failsafe' in case of rogue/bad/buggy code that would lead to never-ending estimate increases. If `at_max` is reached, an RPC request is considered 'broken' and should time out. Setting at_max to 0 causes adaptive timeouts to be disabled and the old fixed-timeout method (`obd_timeout`) to be used. This is the default value in Lustre 1.6.5. **NOTE**: It is possible that slow hardware might validly cause the service estimate to increase beyond the default value of `at_max`. In this case, you should increase `at_max` to the maximum time you are willing to wait for an RPC completion.
at_history	Sets a time period (in seconds) within which adaptive timeouts remember the slowest event that occurred. Default value is 600.

Parameter	Description
at_early_margin	Sets how far before the deadline Lustre sends an early reply. Default value is 5*.
at_extra	Sets the incremental amount of time that a server asks for, with each early reply. The server does not know how much time the RPC will take, so it asks for a fixed value. Default value is 30†. When a server finds a queued request about to time out (and needs to send an early reply out), the server adds the `at_extra` value. If the time expires, the Lustre client enters recovery status and reconnects to restore it to normal status. If you see multiple early replies for the same RPC asking for multiple 30-second increases, change the `at_extra` value to a larger number to cut down on early replies sent and, therefore, network load.
ldlm_enqueue_min	Sets the minimum lock enqueue time. Default value is 100. The `ldlm_enqueue` time is the maximum of the measured enqueue estimate (influenced by `at_min` and `at_max` parameters), multiplied by a weighting factor, and the `ldlm_enqueue_min` setting. LDLM lock enqueues were based on the `obd_timeout` value; now they have a dedicated minimum value. Lock enqueues increase as the measured enqueue times increase (similar to adaptive timeouts).

* This default was chosen as a reasonable time in which to send a reply from the point at which it was sent.

† This default was chosen as a balance between sending too many early replies for the same RPC and overestimating the actual completion time.

In Lustre 1.8, adaptive timeouts are enabled, by default. To disable adaptive timeouts, at run time, set `at_max` to 0. On the MGS, run:

```
$ lctl conf_param <fsname>.sys.at_max=0
```

Note – Changing adaptive timeouts status at runtime may cause transient timeout, reconnect, recovery, etc.

21.1.3.2 Interpreting Adaptive Timeouts Information

Adaptive timeouts information can be read from `/proc/fs/lustre/*/timeouts` files (for each service and client) or with the `lctl` command.

This is an example from the `/proc/fs/lustre/*/timeouts` files:

```
cfs21:~# cat /proc/fs/lustre/ost/OSS/ost_io/timeouts
```

This is an example using the `lctl` command:

```
$ lctl get_param -n ost.*.ost_io.timeouts
```

This is the sample output:

```
service : cur 33  worst 34 (at 1193427052, 0d0h26m40s ago) 1 1 33 2
```

The `ost_io` service on this node is currently reporting an estimate of 33 seconds. The worst RPC service time was 34 seconds, and it happened 26 minutes ago.

The output also provides a history of service times. In the example, there are 4 "bins" of `adaptive_timeout_history`, with the maximum RPC time in each bin reported. In 0-150 seconds, the maximum RPC time was 1, with the same result in 150-300 seconds. From 300-450 seconds, the worst (maximum) RPC time was 33 seconds, and from 450-600s the worst time was 2 seconds. The current estimated service time is the maximum value of the 4 bins (33 seconds in this example).

Service times (as reported by the servers) are also tracked in the client OBDs:

```
cfs21:# lctl get_param osc.*.timeouts
last reply : 1193428639, 0d0h00m00s ago
network    : cur   1  worst   2 (at 1193427053, 0d0h26m26s ago)   1   1   1   1
portal 6   : cur  33  worst  34 (at 1193427052, 0d0h26m27s ago)  33  33  33   2
portal 28  : cur   1  worst   1 (at 1193426141, 0d0h41m38s ago)   1   1   1   1
portal 7   : cur   1  worst   1 (at 1193426141, 0d0h41m38s ago)   1   0   1   1
portal 17  : cur   1  worst   1 (at 1193426177, 0d0h41m02s ago)   1   0   0   1
```

In this case, RPCs to portal 6, the OST_IO_PORTAL (see `lustre/include/lustre/lustre_idl.h`), shows the history of what the `ost_io` portal has reported as the service estimate.

Server statistic files also show the range of estimates in the normal min/max/sum/sumsq manner.

```
cfs21:~# lctl get_param mdt.*.mdt.stats
...
req_timeout               6 samples [sec] 1 10 15 105
...
```

21.1.4 LNET Information

This section describes `/proc` entries for LNET information.

/proc/sys/lnet/peers

Shows all NIDs known to this node and also gives information on the queue state.

```
# cat /proc/sys/lnet/peers
nid              refs   state  max    rtr    min    tx     min queue
0@lo             1      ~rtr   0      0      0      0      0   0
192.168.10.35@tcp1      ~rtr   8      8      8      8      6   0
192.168.10.36@tcp1      ~rtr   8      8      8      8      6   0
192.168.10.37@tcp1      ~rtr   8      8      8      8      6   0
```

The fields are explained below:

Field	Description
refs	A reference count (principally used for debugging)
state	Only valid to refer to routers. Possible values: • ~ rtr (indicates this node is not a router) • up/down (indicates this node is a router) • auto_fail must be enabled
max	Maximum number of concurrent sends from this peer
rtr	Routing buffer credits.
min	Minimum routing buffer credits seen.
tx	Send credits.
min	Minimum send credits seen.
queue	Total bytes in active/queued sends.

Credits work like a semaphore. At start they are initialized to allow a certain number of operations (8 in this example). LNET keeps a track of the minimum value so that you can see how congested a resource was.

If `rtr/tx` is less than `max`, there are operations in progress. The number of operations is equal to `rtr` or `tx` subtracted from `max`.

If `rtr/tx` is greater that `max`, there are operations blocking.

LNET also limits concurrent sends and router buffers allocated to a single peer so that no peer can occupy all these resources.

/proc/sys/lnet/nis

```
# cat /proc/sys/lnet/nis
nid                  refs   peer   max    tx     min
0@lo                 3      0      0      0      0
192.168.10.34@tcp    4      8      256    256    252
```

Shows the current queue health on this node. The fields are explained below:

Field	Description
nid	Network interface
refs	Internal reference counter
peer	Number of peer-to-peer send credits on this NID. Credits are used to size buffer pools
max	Total number of send credits on this NID.
tx	Current number of send credits available on this NID.
min	Lowest number of send credits available on this NID.
queue	Total bytes in active/queued sends.

Subtracting `max - tx` yields the number of sends currently active. A large or increasing number of active sends may indicate a problem.

```
# cat /proc/sys/lnet/nis
nid                  refs   peer   max    tx     min
0@lo                 2      0      0      0      0
10.67.73.173@tcp     4      8      256    256    253
```

21.1.5 Free Space Distribution

Free-space stripe weighting, as set, gives a priority of "0" to free space (versus trying to place the stripes "widely" -- nicely distributed across OSSs and OSTs to maximize network balancing). To adjust this priority (as a percentage), use the `qos_prio_free` proc tunable:

```
$ cat /proc/fs/lustre/lov/<fsname>-mdtlov/qos_prio_free
```

Currently, the default is 90%. You can permanently set this value by running this command on the MGS:

```
$ lctl conf_param <fsname>-MDT0000.lov.qos_prio_free=90
```

Setting the priority to 100% means that OSS distribution does not count in the weighting, but the stripe assignment is still done via weighting. If OST 2 has twice as much free space as OST 1, it is twice as likely to be used, but it is NOT guaranteed to be used.

Also note that free-space stripe weighting does not activate until two OSTs are imbalanced by more than 20%. Until then, a faster round-robin stripe allocator is used. (The new round-robin order also maximizes network balancing.)

21.1.5.1 Managing Stripe Allocation

The MDS uses two methods to manage stripe allocation and determine which OSTs to use for file object storage:

- **QOS**

  Quality of Service (QOS) considers an OST's available blocks, speed, and the number of existing objects, etc. Using these criteria, the MDS selects OSTs with more free space more often than OSTs with less free space.

- **RR**

  Round-Robin (RR) allocates objects evenly across all OSTs. The RR stripe allocator is faster than QOS, and used often because it distributes space usage/load best in most situations, maximizing network balancing and improving performance.

Whether QOS or RR is used depends on the setting of the `qos_threshold_rr` proc tunable. The `qos_threshold_rr` variable specifies a percentage threshold where the use of QOS or RR becomes more/less likely. The `qos_threshold_rr` tunable can be set as an integer, from 0 to 100, and results in this stripe allocation behavior:

- If `qos_threshold_rr` is set to 0, then QOS is always used
- If `qos_threshold_rr` is set to 100, then RR is always used
- The larger the `qos_threshold_rr` setting, the greater the possibility that RR is used instead of QOS

21.2 Lustre I/O Tunables

The section describes I/O tunables.

/proc/fs/lustre/llite/<fsname>-<uid>/max_cache_mb

```
# cat /proc/fs/lustre/llite/lustre-ce63ca00/max_cached_mb 128
```

This tunable is the maximum amount of inactive data cached by the client (default is 3/4 of RAM).

21.2.1 Client I/O RPC Stream Tunables

The Lustre engine always attempts to pack an optimal amount of data into each I/O RPC and attempts to keep a consistent number of issued RPCs in progress at a time. Lustre exposes several tuning variables to adjust behavior according to network conditions and cluster size. Each OSC has its own tree of these tunables. For example:

```
$ ls -d /proc/fs/lustre/osc/OSC_client_ost1_MNT_client_2 /localhost
/proc/fs/lustre/osc/OSC_uml0_ost1_MNT_localhost
/proc/fs/lustre/osc/OSC_uml0_ost2_MNT_localhost
/proc/fs/lustre/osc/OSC_uml0_ost3_MNT_localhost
$ ls /proc/fs/lustre/osc/OSC_uml0_ost1_MNT_localhost
blocksizefilesfreemax_dirty_mb ost_server_uuid stats
```

... and so on.

RPC stream tunables are described below.

/proc/fs/lustre/osc/<object name>/max_dirty_mb

This tunable controls how many MBs of dirty data can be written and queued up in the OSC. POSIX file writes that are cached contribute to this count. When the limit is reached, additional writes stall until previously-cached writes are written to the server. This may be changed by writing a single ASCII integer to the file. Only values between 0 and 512 are allowable. If 0 is given, no writes are cached. Performance suffers noticeably unless you use large writes (1 MB or more).

/proc/fs/lustre/osc/<object name>/cur_dirty_bytes

This tunable is a read-only value that returns the current amount of bytes written and cached on this OSC.

/proc/fs/lustre/osc/<object name>/max_pages_per_rpc

This tunable is the maximum number of pages that will undergo I/O in a single RPC to the OST. The minimum is a single page and the maximum for this setting is platform dependent (256 for i386/x86_64, possibly less for ia64/PPC with larger PAGE_SIZE), though generally amounts to a total of 1 MB in the RPC.

/proc/fs/lustre/osc/<object name>/max_rpcs_in_flight

This tunable is the maximum number of concurrent RPCs in flight from an OSC to its OST. If the OSC tries to initiate an RPC but finds that it already has the same number of RPCs outstanding, it will wait to issue further RPCs until some complete. The minimum setting is 1 and maximum setting is 32. If you are looking to improve small file I/O performance, increase the max_rpcs_in_flight value.

To maximize performance, the value for `max_dirty_mb` is recommended to be 4 * max_pages_per_rpc * `max_rpcs_in_flight`.

Note – The *<object name>* varies depending on the specific Lustre configuration. For *<object name>* examples, refer to the sample command output.

21.2.2 Watching the Client RPC Stream

The same directory contains a `rpc_stats` file with a histogram showing the composition of previous RPCs. The histogram can be cleared by writing any value into the `rpc_stats` file.

```
# cat /proc/fs/lustre/osc/spfs-OST0000-osc-c45f9c00/rpc_stats
snapshot_time:         1174867307.156604 (secs.usecs)
read RPCs in flight:   0
write RPCs in flight:  0
pending write pages:   0
pending read pages:    0
                    read                    write
pages per rpc       rpcs   %  cum %   |  rpcs   %  cum %
1:                  0      0  0       |  0      0  0

                    read                    write
rpcs in flight      rpcs   %  cum %   |  rpcs   %  cum %
0:                  0      0  0       |  0      0  0

                    read                    write
offset              rpcs   %  cum %   |  rpcs   %  cum %
0:                  0      0  0       |  0      0  0
```

Where:

Field	Description
{read,write} RPCs in flight	Number of read/write RPCs issued by the OSC, but not complete at the time of the snapshot. This value should always be less than or equal to `max_rpcs_in_flight`.
pending {read,write} pages	Number of pending read/write pages that have been queued for I/O in the OSC.

Field	Description
pages per RPC	When an RPC is sent, the number of pages it consists of is recorded (in order). A single page RPC increments the 0: row.
RPCs in flight	When an RPC is sent, the number of other RPCs that are pending is recorded. When the first RPC is sent, the 0: row is incremented. If the first RPC is sent while another is pending, the 1: row is incremented and so on. As each RPC *completes*, the number of pending RPCs is not tabulated. This table is a good way to visualize the concurrency of the RPC stream. Ideally, you will see a large clump around the `max_rpcs_in_flight` value, which shows that the network is being kept busy.
offset	

21.2.3 Client Read-Write Offset Survey

The `offset_stats` parameter maintains statistics for occurrences where a series of read or write calls from a process did not access the next sequential location. The offset field is reset to 0 (zero) whenever a different file is read/written.

Read/write offset statistics are off, by default. The statistics can be activated by writing anything into the `offset_stats` file.

Example:

```
# cat /proc/fs/lustre/llite/lustre-f57dee00/rw_offset_stats
snapshot_time: 1155748884.591028 (secs.usecs)
R/W  PID  RANGE STARTRANGE ENDSMALLEST EXTENTLARGEST EXTENTOFFSET
R    8385 0         128       128       128          0
R    8385 0         224       224       224          -128
W    8385 0         250       50        100          0
W    8385 100       1110      10        500          -150
W    8384 0         5233      5233      5233         0
R    8385 500       600       100       100          -610
```

Where:

Field	Description
R/W	Whether the non-sequential call was a read or write
PID	Process ID which made the read/write call.
Range Start/Range End	Range in which the read/write calls were sequential.
Smallest Extent	Smallest extent (single read/write) in the corresponding range.
Largest Extent	Largest extent (single read/write) in the corresponding range.
Offset	Difference from the previous range end to the current range start. For example, `Smallest-Extent` indicates that the writes in the range 100 to 1110 were sequential, with a minimum write of 10 and a maximum write of 500. This range was started with an offset of -150. That means this is the difference between the last entry's range-end and this entry's range-start for the same file. The `rw_offset_stats` file can be cleared by writing to it: `echo > /proc/fs/lustre/llite/lustre-f57dee00/rw_offset_stats`

21.2.4 Client Read-Write Extents Survey

Client-Based I/O Extent Size Survey

The `rw_extent_stats` histogram in the llite directory shows you the statistics for the sizes of the read-write I/O extents. This file does not maintain the per-process statistics.

Example:

```
$ cat /proc/fs/lustre/llite/lustre-ee5af200/extents_stats
snapshot_time:      1213828728.348516 (secs.usecs)
                    read            |          write
extents           calls  %   cum%   |      calls  %      cum%

0K - 4K :         0      0   0      |      2      2      2
4K - 8K :         0      0   0      |      0      0      2
8K - 16K :        0      0   0      |      0      0      2
16K - 32K :       0      0   0      |      20     23     26
32K - 64K :       0      0   0      |      0      0      26
64K - 128K :      0      0   0      |      51     60     86
128K - 256K :     0      0   0      |      0      0      86
256K - 512K :     0      0   0      |      0      0      86
512K - 1024K :    0      0   0      |      0      0      86
1M - 2M :         0      0   0      |      11     13     100
```

The file can be cleared by issuing the following command:

```
$ echo > cat /proc/fs/lustre/llite/lustre-ee5af200/extents_stats
```

Per-Process Client I/O Statistics

The extents_stats_per_process file maintains the I/O extent size statistics on a per-process basis. So you can track the per-process statistics for the last MAX_PER_PROCESS_HIST processes.

Example:

```
$ cat /proc/fs/lustre/llite/lustre-ee5af200/extents_stats_per_process
snapshot_time:      1213828762.204440 (secs.usecs)
                    read          |         write
extents          calls  %   cum%   |      calls  %   cum%

PID: 11488
   0K - 4K :     0      0    0     |      0      0      0
   4K - 8K :     0      0    0     |      0      0      0
   8K - 16K :    0      0    0     |      0      0      0
   16K - 32K :   0      0    0     |      0      0      0
   32K - 64K :   0      0    0     |      0      0      0
   64K - 128K :  0      0    0     |      0      0      0
   128K - 256K : 0      0    0     |      0      0      0
   256K - 512K : 0      0    0     |      0      0      0
   512K - 1024K :0      0    0     |      0      0      0
   1M - 2M :     0      0    0     |      10     100    100

PID: 11491
   0K - 4K :     0      0    0     |      0      0      0
   4K - 8K :     0      0    0     |      0      0      0
   8K - 16K :    0      0    0     |      0      0      0
   16K - 32K :   0      0    0     |      20     100    100

PID: 11424
   0K - 4K :     0      0    0     |      0      0      0
   4K - 8K :     0      0    0     |      0      0      0
   8K - 16K :    0      0    0     |      0      0      0
   16K - 32K :   0      0    0     |      0      0      0
   32K - 64K :   0      0    0     |      0      0      0
   64K - 128K :  0      0    0     |      16     100    100

PID: 11426
   0K - 4K :     0      0    0     |      1      100    100

PID: 11429
   0K - 4K :     0      0    0     |      1      100    100
```

21.2.5 Watching the OST Block I/O Stream

Similarly, there is a `brw_stats` histogram in the obdfilter directory which shows you the statistics for number of I/O requests sent to the disk, their size and whether they are contiguous on the disk or not.

```
cat /proc/fs/lustre/obdfilter/lustre-OST0000/brw_stats
snapshot_time:         1174875636.764630 (secs:usecs)
                              read                write
pages per brw       brws   %  cum %   |  rpcs   %  cum %
1:                  0      0  0       |  0      0  0
                              read                write
discont pages       rpcs   %  cum %   |  rpcs   %  cum %
1:                  0      0  0       |  0      0  0
                              read                write
discont blocks      rpcs   %  cum %   |  rpcs   %  cum %
1:                  0      0  0       |  0      0  0
                              read                write
dio frags           rpcs   %  cum %   |  rpcs   %  cum %
1:                  0      0  0       |  0      0  0
                              read                write
disk ios in flight  rpcs   %  cum %   |  rpcs   %  cum %
1:                  0      0  0       |  0      0  0
                              read                write
io time (1/1000s)   rpcs   %  cum %   |  rpcs   %  cum %
1:                  0      0  0       |  0      0  0
                              read                write
disk io size        rpcs   %  cum %   |  rpcs   %  cum %
1:                  0      0  0       |  0      0  0
                              read                write
```

The fields are explained below:

Field	Description
pages per brw	Number of pages per RPC request, which should match aggregate client `rpc_stats`.
discont pages	Number of discontinuities in the logical file offset of each page in a single RPC.
discont blocks	Number of discontinuities in the physical block allocation in the file system for a single RPC.

For each Lustre service, the following information is provided:

- Number of requests
- Request wait time (avg, min, max and std dev)
- Service idle time (% of elapsed time)

Additionally, data on each Lustre service is provided by service type:

- Number of requests of this type
- Request service time (avg, min, max and std dev)

21.2.6 Using File Readahead and Directory Statahead

Lustre 1.6.5.1 introduced file readahead and directory statahead functionality that read data into memory in anticipation of a process actually requesting the data. File readahead functionality reads file content data into memory. Directory statahead functionality reads metadata into memory. When readahead and/or statahead work well, a data-consuming process finds that the information it needs is available when requested, and it is unnecessary to wait for network I/O.

21.2.6.1 Tuning File Readahead

File readahead is triggered when two or more sequential reads by an application fail to be satisfied by the Linux buffer cache. The size of the initial readahead is 1 MB. Additional readaheads grow linearly, and increment until the readahead cache on the client is full at 40 MB.

/proc/fs/lustre/llite/<fsname>-<uid>/max_read_ahead_mb

This tunable controls the maximum amount of data readahead on a file. Files are read ahead in RPC-sized chunks (1 MB or the size of read() call, if larger) after the second sequential read on a file descriptor. Random reads are done at the size of the read() call only (no readahead). Reads to non-contiguous regions of the file reset the readahead algorithm, and readahead is not triggered again until there are sequential reads again. To disable readahead, set this tunable to 0. The default value is 40 MB.

/proc/fs/lustre/llite/<fsname>-<uid>/max_read_ahead_whole_mb

This tunable controls the maximum size of a file that is read in its entirety, regardless of the size of the read().

21.2.6.2 Tuning Directory Statahead

When the `ls -l` process opens a directory, its process ID is recorded. When the first directory entry is ''stated'' with this recorded process ID, a statahead thread is triggered which stats ahead all of the directory entries, in order. The `ls -l` process can use the stated directory entries directly, improving performance.

/proc/fs/lustre/llite/*/statahead_max

This tunable controls whether directory statahead is enabled and the maximum statahead count. By default, statahead is active.

To disable statahead, set this tunable to:

```
echo 0 > /proc/fs/lustre/llite/*/statahead_max
```

To set the maximum statahead count (`n`), set this tunable to:

```
echo n > /proc/fs/lustre/llite/*/statahead_max
```

The maximum value of `n` is 8192.

/proc/fs/lustre/llite/*/statahead_status

This is a read-only interface that indicates the current statahead status.

21.2.7 OSS Read Cache

Lustre 1.8 introduces the OSS read cache feature, which provides read-only caching of data on an OSS. This functionality uses the regular Linux page cache to store the data. Just like caching from a regular filesytem in Linux, OSS read cache uses as much physical memory as is allocated.

OSS read cache improves Lustre performance in these situations:

- Many clients are accessing the same data set (as in HPC applications and when diskless clients boot from Lustre)
- One client is storing data while another client is reading it (essentially exchanging data via the OST)
- A client has very limited caching of its own

OSS read cache offers these benefits:

- Allows OSTs to cache read data more frequently
- Improves repeated reads to match network speeds instead of disk speeds
- Provides the building blocks for OST write cache (small-write aggregation)

21.2.7.1 Using OSS Read Cache

OSS read cache is implemented on the OSS, and does not require any special support on the client side. Since OSS read cache uses the memory available in the Linux page cache, you should use I/O patterns to determine the appropriate amount of memory for the cache; if the data is mostly reads, then more cache is required than for writes.

OSS read cache is enabled, by default, and managed by the following tunables:

- `read_cache_enable` controls whether data read from disk during a read request is kept in memory and available for later read requests for the same data, without having to re-read it from disk. By default, read cache is enabled (read_cache_enable = 1).

  When the OSS receives a read request from a client, it reads data from disk into its memory and sends the data as a reply to the requests. If read cache is enabled, this data stays in memory after the client's request is finished, and the OSS skips reading data from disk when subsequent read requests for the same are received. The read cache is managed by the Linux kernel globally across all OSTs on that OSS, and the least recently used cache pages will be dropped from memory when the amount of free memory is running low.

  If read cache is disabled (`read_cache_enable` = 0), then the OSS will discard the data after the client's read requests are serviced and, for subsequent read requests, the OSS must read the data from disk.

To disable read cache on all OSTs of an OSS, run:

```
root@oss1# lctl set_param obdfilter.*.read_cache_enable=0
```

To re-enable read cache on one OST, run:

```
root@oss1# lctl set_param obdfilter.{OST_name}.read_cache_enable=1
```

To check if read cache is enabled on all OSTs on an OSS, run:

```
root@oss1# lctl get_param obdfilter.*.read_cache_enable
```

- `writethrough_cache_enable` controls whether data sent to the OSS as a write request is kept in the read cache and available for later reads, or if it is discarded from cache when the write is completed. By default, writethrough cache is enabled (`writethrough_cache_enable` = 1).

  When the OSS receives write requests from a client, it receives data from the client into its memory and writes the data to disk. If writethrough cache is enabled, this data stays in memory after the write request is completed, allowing the OSS to skip reading this data from disk if a later read request, or partial-page write request, for the same data is received.

  If writethrough cache is disabled (`writethrough_cache_enabled` = 0), then the OSS discards the data after the client's write request is completed, and for subsequent read request, or partial-page write request, the OSS must re-read the data from disk.

  Enabling writethrough cache is advisable if clients are doing small or unaligned writes that would cause partial-page updates, or if the files written by one node are immediately being accessed by other nodes. Some examples where this might be useful include producer-consumer I/O models or shared-file writes with a different node doing I/O not aligned on 4096-byte boundaries. Disabling writethrough cache is advisable in the case where files are mostly written to the file system but are not re-read within a short time period, or files are only written and re-read by the same node, regardless of whether the I/O is aligned or not.

  To disable writethrough cache on all OSTs of an OSS, run:

  ```
  root@oss1# lctl set_param obdfilter.*.writethrough_cache_enable=0
  ```

  To re-enable writethrough cache on one OST, run:

  ```
  root@oss1# lctl set_param \
  obdfilter.{OST_name}.writethrough_cache_enable=1
  ```

  To check if writethrough cache is

  ```
  root@oss1# lctl set_param obdfilter.*.writethrough_cache_enable=1
  ```

- `readcache_max_filesize` controls the maximum size of a file that both the read cache and writethrough cache will try to keep in memory. Files larger than `readcache_max_filesize` will not be kept in cache for either reads or writes.

  This can be very useful for workloads where relatively small files are repeatedly accessed by many clients, such as job startup files, executables, log files, etc., but large files are read or written only once. By not putting the larger files into the cache, it is much more likely that more of the smaller files will remain in cache for a longer time.

  When setting `readcache_max_filesize`, the input value can be specified in bytes, or can have a suffix to indicate other binary units such as **K**ilobytes, **M**egabytes, **G**igabytes, **T**erabytes, or **P**etabytes.

  To limit the maximum cached file size to 32MB on all OSTs of an OSS, run:

  ```
  root@oss1# lctl set_param obdfilter.*.readcache_max_filesize=32M
  ```

  To disable the maximum cached file size on an OST, run:

  ```
  root@oss1# lctl set_param \
  obdfilter.{OST_name}.readcache_max_filesize=-1
  ```

  To check the current maximum cached file size on all OSTs of an OSS, run:

  ```
  root@oss1# lctl get_param obdfilter.*.readcache_max_filesize
  ```

21.2.8 mballoc History

/proc/fs/ldiskfs/sda/mb_history

Multi-Block-Allocate (mballoc), enables Lustre to ask ext3 to allocate multiple blocks with a single request to the block allocator. Typically, an ext3 file system allocates only one block per time. Each mballoc-enabled partition has this file. This is sample output:

```
pid   inode     goal        result      found   grpscr  \   merge   tailbroken
2838  139267    17/12288/1  17/12288/1  1       0   0   \   M       1   8192
2838  139267    17/12289/1  17/12289/1  1       0   0   \   M       0   0
2838  139267    17/12290/1  17/12290/1  1       0   0   \   M       1   2
2838  24577     3/12288/1   3/12288/1   1       0   0   \   M       1   8192
2838  24578     3/12288/1   3/771/1     1       1   1   \           0   0
2838  32769     4/12288/1   4/12288/1   1       0   0   \   M       1   8192
2838  32770     4/12288/1   4/12289/1   13      1   1   \           0   0
2838  32771     4/12288/1   5/771/1     26      2   1   \           0   0
2838  32772     4/12288/1   5/896/1     31      2   1   \           1   128
2838  32773     4/12288/1   5/897/1     31      2   1   \           0   0
2828  32774     4/12288/1   5/898/1     31      2   1   \           1   2
2838  32775     4/12288/1   5/899/1     31      2   1   \           0   0
2838  32776     4/12288/1   5/900/1     31      2   1   \           1   4
2838  32777     4/12288/1   5/901/1     31      2   1   \           0   0
2838  32778     4/12288/1   5/902/1     31      2   1   \           1   2
```

The parameters are described below:

Parameter	Description
pid	Process that made the allocation.
inode	inode number allocated blocks
goal	Initial request that came to mballoc (group/block-in-group/number-of-blocks)
result	What mballoc actually found for this request.
found	Number of free chunks mballoc found and measured before the final decision.
grps	Number of groups mballoc scanned to satisfy the request.
cr	Stage at which mballoc found the result: **0** - best in terms of resource allocation. The request was 1MB or larger and was satisfied directly via the kernel buddy allocator. **1** - regular stage (good at resource consumption) **2** - fs is quite fragmented (not that bad at resource consumption) **3** - fs is very fragmented (worst at resource consumption)
queue	Total bytes in active/queued sends.

Parameter	Description
merge	Whether the request hit the goal. This is good as extents code can now merge new blocks to existing extent, eliminating the need for extents tree growth.
tail	Number of blocks left free after the allocation breaks large free chunks.
broken	How large the broken chunk was.

Most customers are probably interested in `found/cr`. If `cr` is 0 1 and `found` is less than 100, then mballoc is doing quite well.

Also, number-of-blocks-in-request (third number in the goal triple) can tell the number of blocks requested by the obdfilter. If the obdfilter is doing a lot of small requests (just few blocks), then either the client is processing input/output to a lot of small files, or something may be wrong with the client (because it is better if client sends large input/output requests). This can be investigated with the OSC `rpc_stats` or OST `brw_stats` mentioned above.

Number of groups scanned (grps column) should be small. If it reaches a few dozen often, then either your disk file system is pretty fragmented or mballoc is doing something wrong in the group selection part.

21.2.9 mballoc3 Tunables

Lustre version 1.6.1 and later includes mballoc3, which was built on top of mballoc2. By default, mballoc3 is enabled, and adds these features:

- Pre-allocation for single files (helps to resist fragmentation)
- Pre-allocation for a group of files (helps to pack small files into large, contiguous chunks)
- Stream allocation (helps to decrease the seek rate)

The following mballoc3 tunables are available:

Field	Description
stats	Enables/disables the collection of statistics. Collected statistics can be found in /proc/fs/ldiskfs2/<dev>/mb_history.
max_to_scan	Maximum number of free chunks that mballoc finds before a final decision to avoid livelock.
min_to_scan	Minimum number of free chunks that mballoc finds before a final decision. This is useful for a very small request, to resist fragmentation of big free chunks.
order2_req	For requests equal to 2^N (where N >= order2_req), a very fast search via buddy structures is used.
stream_req	Requests smaller or equal to this value are packed together to form large write I/Os.

The following tunables, providing more control over allocation policy, will be available in the next version:

Field	Description
stats	Enables/disables the collection of statistics. Collected statistics can be found in /proc/fs/ldiskfs2/<dev>/mb_history.
max_to_scan	Maximum number of free chunks that mballoc finds before a final decision to avoid livelock.
min_to_scan	Minimum number of free chunks that mballoc finds before a final decision. This is useful for a very small request, to resist fragmentation of big free chunks.
order2_req	For requests equal to 2^N (where N >= order2_req), a very fast search via buddy structures is used.
small_req **large_req**	All requests are divided into 3 categories: < `small_req` (packed together to form large, aggregated requests) < `large_req` (allocated mostly in linearly) > `large_req` (very large requests so the arm seek does not matter) The idea is that we try to pack small requests to form large requests, and then place all large requests (including compound from the small ones) close to one another, causing as few arm seeks as possible.
prealloc_table	The amount of space to preallocate depends on the current file size. The idea is that for small files we do not need 1 MB preallocations and for large files, 1 MB preallocations are not large enough; it is better to preallocate 4 MB.
group_prealloc	The amount of space preallocated for small requests to be grouped.

21.2.10 Locking

/proc/fs/lustre/ldlm/ldlm/namespaces/<OSC name|MDCname>/lru_size

The `lru_size` parameter is used to control the number of client-side locks in an LRU queue. LRU size is dynamic, based on load. This optimizes the number of locks available to nodes that have different workloads (e.g., login/build nodes vs. compute nodes vs. backup nodes).

The total number of locks available is a function of the server's RAM. The default limit is 50 locks/1 MB of RAM. If there is too much memory pressure, then the LRU size is shrunk. The number of locks on the server is limited to {number of OST/MDT on node} * {number of clients} * {client lru_size}.

- To enable automatic LRU sizing, set the `lru_size` parameter to 0. In this case, the `lru_size` parameter shows the current number of locks being used on the export. (In Lustre 1.6.5.1 and later, LRU sizing is enabled, by default.)
- To specify a maximum number of locks, set the `lru_size` parameter to a value > 0 (former numbers are okay, 100 * CPU_NR). We recommend that you only increase the LRU size on a few login nodes where users access the file system interactively.

To clear the LRU on a single client, and as a result flush client cache, without changing the `lru_size` value:

```
$ lctl set_param ldlm.namespaces.<osc_name|mdc_name>.lru_size=clear
```

If you shrink the LRU size below the number of existing unused locks, then the unused locks are canceled immediately. Use `echo clear` to cancel all locks without changing the value.

Note – Currently, the `lru_size` parameter can only be set temporarily with `lctl set_param`; it cannot be set permanently.

To disable LRU sizing, run this command on the Lustre clients:

```
$ lctl set_param ldlm.namespaces.*osc*.lru_size=$((NR_CPU*100))
```

Replace `NR_CPU` value with the number of CPUs on the node.

To determine the number of locks being granted:

```
$ lctl get_param ldlm.namespaces.*.pool.limit
```

21.2.11 Setting MDS and OSS Thread Counts

In Lustre 1.8 and later, MDS and OSS thread counts (minimum and maximum) can be set via the `{min,max}_thread_count` tunable. For each service, a new `/proc/fs/lustre/{service}/*/thread_{min,max,started}` entry is created. The tunable, `{service}.thread_{min,max,started}`, can be used to set the minimum and maximum thread counts or get the current number of running threads for the following services.

Service	Description
mdt.MDS.mds	normal metadata ops
mdt.MDS.mds_readpage	metadata readdir
mdt.MDS.mds_setattr	metadata setattr
ost.OSS.ost	normal data
ost.OSS.ost_io	bulk data IO
ost.OSS.ost_create	OST object pre-creation service
ldlm.services.ldlm_canceld	DLM lock cancel
ldlm.services.ldlm_cbd	DLM lock grant

- To temporarily set this tunable, run:

```
# lctl {get,set}_param {service}.thread_{min,max,started}
```

- To permanently set this tunable, run:

```
# lctl conf_param {service}.thread_{min,max,started}
```

The following examples show how to set thread counts and get the number of running threads for the `ost_io` service.

- To get the number of running threads, run:

```
# lctl get_param ost.OSS.ost_io.threads_started
```

  The command output will be similar to this:

```
ost.OSS.ost_io.threads_started=128
```

- To set the maximum number of threads (512), run:

```
# lctl get_param ost.OSS.ost_io.threads_max
```

  The command output will be:

```
ost.OSS.ost_io.threads_max=512
```

- To set the maximum thread count to 256 instead of 512 (to avoid overloading the storage or for an array with requests), run:

  ```
  # lctl set_param ost.OSS.ost_io.threads_max=256
  ```

  The command output will be:

  ```
  ost.OSS.ost_io.threads_max=256
  ```

- To check if the new `threads_max` setting is active, run:

  ```
  # lctl get_param ost.OSS.ost_io.threads_max
  ```

  The command output will be similar to this:

  ```
  ost.OSS.ost_io.threads_max=256
  ```

Note – Currently, the maximum thread count setting is advisory because Lustre does not reduce the number of service threads in use, even if that number exceeds the `threads_max` value. Lustre does not stop service threads once they are started.

21.3 Debug Support

/proc/sys/lnet/debug

By default, Lustre generates a detailed log of all operations to aid in debugging. The level of debugging can affect the performance or speed you achieve with Lustre. Therefore, it is useful to reduce this overhead by turning down the debug level[1] to improve performance. Raise the debug level when you need to collect the logs for debugging problems. The debugging mask can be set with "symbolic names" instead of the numerical values that were used in prior releases. The new symbolic format is shown in the examples below.

Note – All of the commands below must be run as root; note the # nomenclature.

To verify the debug level used by examining the sysctl that controls debugging, run:

```
# sysctl lnet.debug
lnet.debug = ioctl neterror warning error emerg ha config console
```

To turn off debugging (except for network error debugging), run this command on all concerned nodes:

```
# sysctl -w lnet.debug="neterror"
lnet.debug = neterror
```

To turn off debugging completely, run this command on all concerned nodes:

```
# sysctl -w lnet.debug=0
lnet.debug = 0
```

To set an appropriate debug level for a production environment, run:

```
# sysctl -w lnet.debug="warning dlmtrace error emerg ha rpctrace
vfstrace"
lnet.debug = warning dlmtrace error emerg ha rpctrace vfstrace
```

The flags above collect enough high-level information to aid debugging, but they do not cause any serious performance impact.

To clear all flags and set new ones, run:

```
# sysctl -w lnet.debug="warning"
lnet.debug = warning
```

1. This controls the level of Lustre debugging kept in the internal log buffer. It does not alter the level of debugging that goes to syslog.

To add new flags to existing ones, prefix them with a "+":

```
# sysctl -w lnet.debug="+neterror +ha"
lnet.debug = +neterror +ha

# sysctl lnet.debug
lnet.debug = neterror warning ha
```

To remove flags, prefix them with a "-":

```
# sysctl -w lnet.debug="-ha"
lnet.debug = -ha

# sysctl lnet.debug
lnet.debug = neterror warning
```

You can verify and change the debug level using the /proc interface in Lustre. To use the flags with /proc, run:

```
# cat /proc/sys/lnet/debug
neterror warning

# echo "+ha" > /proc/sys/lnet/debug

# cat /proc/sys/lnet/debug
neterror warning ha

# echo "-warning" > /proc/sys/lnet/debug

# cat /proc/sys/lnet/debug
neterror ha
```

/proc/sys/lnet/subsystem_debug

This controls the debug logs[3] for subsystems (see S_* definitions).

/proc/sys/lnet/debug_path

This indicates the location where debugging symbols should be stored for gdb. The default is set to `/r/tmp/lustre-log-localhost.localdomain`.

These values can also be set via `sysctl -w lnet.debug={value}`

Note – The above entries only exist when Lustre has already been loaded.

/proc/sys/lnet/panic_on_lbug

This causes Lustre to call "panic" when it detects an internal problem (an LBUG); panic crashes the node. This is particularly useful when a kernel crash dump utility is configured. The crash dump is triggered when the internal inconsistency is detected by Lustre.

/proc/sys/lnet/upcall

This allows you to specify the path to the binary which will be invoked when an LBUG is encountered. This binary is called with four parameters. The first one is the string "LBUG". The second one is the file where the LBUG occurred. The third one is the function name. The fourth one is the line number in the file.

21.3.1 RPC Information for Other OBD Devices

Some OBD devices maintain a count of the number of RPC events that they process. Sometimes these events are more specific to operations of the device, like *llite*, than actual raw RPC counts.

```
$ find /proc/fs/lustre/ -name stats
/proc/fs/lustre/osc/lustre-OST0001-osc-ce63ca00/stats
/proc/fs/lustre/osc/lustre-OST0000-osc-ce63ca00/stats
/proc/fs/lustre/osc/lustre-OST0001-osc/stats
/proc/fs/lustre/osc/lustre-OST0000-osc/stats
/proc/fs/lustre/mdt/MDS/mds_readpage/stats
/proc/fs/lustre/mdt/MDS/mds_setattr/stats
/proc/fs/lustre/mdt/MDS/mds/stats
/proc/fs/lustre/mds/lustre-MDT0000/exports/ab206805-0630-6647-8543-d
24265c91a3d/stats
/proc/fs/lustre/mds/lustre-MDT0000/exports/08ac6584-6c4a-3536-2c6d-b
36cf9cbdaa0/stats
/proc/fs/lustre/mds/lustre-MDT0000/stats
/proc/fs/lustre/ldlm/services/ldlm_canceld/stats
/proc/fs/lustre/ldlm/services/ldlm_cbd/stats
/proc/fs/lustre/llite/lustre-ce63ca00/stats
```

21.3.1.1 Interpreting OST Statistics

The OST .../stats files can be used to track client statistics (client activity) for each OST. It is possible to get a periodic dump of values from these file (for example, every 10 seconds), that show the RPC rates (similar to iostat) by using the llstat.pl tool:

```
# llstat /proc/fs/lustre/osc/lustre-OST0000-osc/stats

/usr/bin/llstat: STATS on 09/14/07
/proc/fs/lustre/osc/lustre-OST0000-osc/stats on 192.168.10.34@tcp
snapshot_time           1189732762.835363
ost_create              1
ost_get_info            1
ost_connect             1
ost_set_info            1
obd_ping                212
```

To clear the statistics, give the -c option to llstat.pl. To specify how frequently the statistics should be cleared (in seconds), use an integer for the -i option. This is sample output with -c and -i10 options used, providing statistics every 10s):

```
$ llstat -c -i10 /proc/fs/lustre/ost/OSS/ost_io/stats

/usr/bin/llstat: STATS on 06/06/07 /proc/fs/lustre/ost/OSS/ost_io/ stats on
192.168.16.35@tcp
snapshot_time                 1181074093.276072

/proc/fs/lustre/ost/OSS/ost_io/stats @ 1181074103.284895
Name        Cur.CountCur.Rate#EventsUnit\  last    min avg     max     stddev
req_waittime8       0       8   [usec]  2078\   34  259.75  868     317.49
req_qdepth  8       0       8   [reqs]  1\      0   0.12    1       0.35
req_active  8       0       8   [reqs]  11\     1   1.38    2       0.52
reqbuf_avail8       0       8   [bufs]  511\    63  63.88   64      0.35
ost_write   8       0       8   [bytes]1697677\72914212209.6238757991874.29

/proc/fs/lustre/ost/OSS/ost_io/stats @ 1181074113.290180
Name        Cur.CountCur.Rate#EventsUnit \     lastmin avg max     stddev
req_waittime31      3       39  [usec]  30011\      34  822.79  12245   2047.71
req_qdepth  31      3       39  [reqs]  0\          0   0.03    1       0.16
req_active  31      3       39  [reqs]  58\         1   1.77    3       0.74
reqbuf_avail31      3       39  [bufs]  1977\       63  63.79   64      0.41
ost_write   30      3       38  [bytes]10284679\15019315325.16910694197776.51

/proc/fs/lustre/ost/OSS/ost_io/stats @ 1181074123.325560
Name        Cur.CountCur.Rate#Events Unit \   last minavgmax     stddev
req_waittime21          2       60  [usec] 14970\ 34784.32122451878.66
req_qdepth  21          2       60  [reqs] 0\ 0    0.02    1   0.13
req_active  21          2       60  [reqs] 33\ 1   1.70    3   0.70
reqbuf_avail21          2       60  [bufs] 1341\ 6363.82   64  0.39
ost_write   21          2       59  [bytes]7648424\ 15019332725.08910694
180397.87
```

Where:

Parameter	Description
Cur. Count	Number of events of each type sent in the last interval (in this example, 10s)
Cur. Rate	Number of events per second in the last interval
#Events	Total number of such events since the system started
Unit	Unit of measurement for that statistic (microseconds, requests, buffers)
last	Average rate of these events (in units/event) for the last interval during which they arrived. For instance, in the above mentioned case of ost_destroy it took an average of 736 microseconds per destroy for the 400 object destroys in the previous 10 seconds.
min	Minimum rate (in units/events) since the service started
avg	Average rate
max	Maximum rate
stddev	Standard deviation (not measured in all cases)

The events common to all services are:

Parameter	Description
req_waittime	Amount of time a request waited in the queue before being handled by an available server thread.
req_qdepth	Number of requests waiting to be handled in the queue for this service.
req_active	Number of requests currently being handled.
reqbuf_avail	Number of unsolicited lnet request buffers for this service.

Some service-specific events of interest are:

Parameter	Description
ldlm_enqueue	Time it takes to enqueue a lock (this includes file open on the MDS)
mds_reint	Time it takes to process an MDS modification record (includes create, mkdir, unlink, rename and setattr)

21.3.1.2 llobdstat

The `llobdstat` utility displays statistics for the activity of a specific OST on an OSS:

`/proc/fs/lustre/<ost_name>/stats`

Use `llobdstat` to monitor changes in statistics over time, and I/O rates for all OSTs on a server. the `llobdstat` utility provides utilization graphs for selectable time-scales.

Usage:

`#llobdstat <ost_name> [<interval>]`

Parameter	Description
ost_name	The OST name under /proc/fs/lustre/obdfilter
interval	Sample interval (in seconds)

Example:

```
llobdstat lustre-OST0000 2
```

21.3.1.3 Interpreting MDT Statistics

The MDT `.../stats` files can be used to track MDT statistics for the MDS. Here is sample output for an MDT stats file:

```
# cat /proc/fs/lustre/mds/*-MDT0000/stats
snapshot_time           1244832003.676892 secs.usecs
open                    2 samples [reqs]
close                   1 samples [reqs]
getxattr                3 samples [reqs]
process_config          1 samples [reqs]
connect                 2 samples [reqs]
disconnect              2 samples [reqs]
statfs                  3 samples [reqs]
setattr                 1 samples [reqs]
getattr                 3 samples [reqs]
llog_init               6 samples [reqs]
notify                  16 samples [reqs]
```

CHAPTER 22

Lustre Monitoring and Troubleshooting

This chapter provides information to troubleshoot Lustre, submit a Lustre bug, and Lustre performance tips. It includes the following sections:

- Monitoring Lustre
- Troubleshooting Lustre
- Reporting a Lustre Bug
- Common Lustre Problems and Performance Tips

22.1 Monitoring Lustre

Several tools are available to monitor a Lustre cluster.

Lustre Monitoring Tool

The Lustre Monitoring Tool (LMT[1]) is a Python-based, distributed system that provides a "top" like display of activity on server-side nodes[2] (MDS, OSS and portals routers) on one or more Lustre file systems. For more information on LMT, including the setup procedure, see:

http://code.google.com/p/lmt/

LMT questions can be directed to:

lmt-discuss@googlegroups.com

1. LMT was developed by Lawrence Livermore National Lab (LLNL) and continues to be maintained by LLNL.
2. Lustre client monitoring is not supported.

Red Hat Cluster Manager

The Red Hat Cluster Manager provides high availability features that are essential for data integrity, application availability and uninterrupted service under various failure conditions. You can use the Cluster Manager to test MDS/OST failure in Lustre clusters.

To use Cluster Manager to test MDS failover, specific hardware is required - a compute node, OSTs and two machines (to act as the active and failover MDSs). The MDS nodes need to be able to see the same shared storage, so you need to prepare a shared disk for the Cluster Manager and the MDSs. Several RPM packages are also required[3], along with certain configuration changes.

For more information on the Cluster Manager (bundled in the Red Hat Cluster Suite), see the Red Hat Cluster Suite. Supporting documentation is available to the Red Hat Cluster Suite Overview.

For more information on installing and configuring Cluster Manager for Lustre failover, and testing MDS failover, see Cluster Manager.

SNMP Monitoring

Lustre has a native SNMP module, which enables you to use various standard SNMP monitoring packages (anything using RRDTool as a backend) to track performance. For more information in installing, building and using the SNMP module, see Lustre SNMP Module.

CollectL

CollectL is another tool that can be used to monitor Lustre. You can run CollectL on a Lustre system that has any combination of MDSs, OSTs and clients. The collected data can be written to a file for continuous logging and played back at a later time. It can also be converted to a format suitable for plotting.

For more information about CollectL, see:

http://collectl.sourceforge.net

Lustre-specific documentation is also available. See:

http://collectl.sourceforge.net/Tutorial-Lustre.html

3. The Lustre Group has made several scripts available for MDS failover testing.

Other Monitoring Options

Another option is to script a simple monitoring solution which looks at various reports from ipconfig, as well as the procfs files generated by Lustre.

22.2 Troubleshooting Lustre

Several resources are available to help use troubleshoot Lustre. This section describes error numbers, error messages and logs.

22.2.1 Error Numbers

Error numbers for Lustre come from the Linux errno.h, and are located in /usr/include/asm/errno.h. Lustre does not use all of the available Linux error numbers. The exact meaning of an error number depends on where it is used. Here is a summary of the basic errors that Lustre users may encounter.

Error Number	Error Name	Description
-1	-EPERM	Permission is denied.
-2	-ENOENT	The requested file or directory does not exist.
-4	-EINTR	The operation was interrupted (usually CTRL-C or a killing process).
-5	-EIO	The operation failed with a read or write error.
-19	-ENODEV	No such device is available. The server stopped or failed over.
-22	-EINVAL	The parameter contains an invalid value.
-28	-ENOSPC	The file system is out-of-space or out of inodes. Use `lfs df` (query the amount of file system space) or `lfs df -i` (query the number of inodes).
-30	-EROFS	The file system is read-only, likely due to a detected error.
-43	-EIDRM	The UID/GID does not match any known UID/GID on the MDS. Update etc/hosts and etc/group on the MDS to add the missing user or group.
-107	-ENOTCONN	The client is not connected to this server.
-110	-ETIMEDOUT	The operation took too long and timed out.

22.2.2 Error Messages

As Lustre code runs on the kernel, single-digit error codes display to the application; these error codes are an indication of the problem. Refer to the kernel console log (dmesg) for all recent kernel messages from that node. On the node, `/var/log/messages` holds a log of all messages for at least the past day.

22.2.3 Lustre Logs

The error message initiates with "LustreError" in the console log and provides a short description of:

- What the problem is
- Which process ID had trouble
- Which server node it was communicating with, and so on.

Lustre logs are dumped to `/proc/sys/lnet/debug_path`.

Collect the first group of messages related to a problem, and any messages that precede "LBUG" or "assertion failure" errors. Messages that mention server nodes (OST or MDS) are specific to that server; you must collect similar messages from the relevant server console logs.

Another Lustre debug log holds information for Lustre action for a short period of time which, in turn, depends on the processes on the node to use Lustre. Use the following command to extract debug logs on each of the nodes, run

```
$ lctl dk <filename>
```

Note – LBUG freezes the thread to allow capture of the panic stack. A system reboot is needed to clear the thread.

22.3 Reporting a Lustre Bug

If, after troubleshooting your Lustre system, you cannot resolve the problem, consider reporting a Lustre bug. To do this, you will need an account on Bugzilla (defect tracking system used for Lustre), and download the Lustre diagnostics tool to run and capture the diagnostics output.

Note – Create a Lustre Bugzilla account. Download the Lustre diagnostics tool and install it on the affected nodes. Make sure you are using the most recent version of the diagnostics tool.

---

1. **Once you have a Lustre Bugzilla account, open a new bug and describe the problem you having.**
2. **Run the Lustre diagnostics tool, using one of the following commands:**

```
# lustre-diagnostics -t <bugzilla bug #>
# lustre-diagnostics.
```

In case you need to use it later, the output of the bug is sent directly to the terminal. Normal file redirection can be used to send the output to a file which you can manually attach to this bug, if necessary.

22.4 Common Lustre Problems and Performance Tips

This section describes common issues encountered with Lustre, as well as tips to improve Lustre performance.

22.4.1 Recovering from an Unavailable OST

One of the most common problems encountered in a Lustre environment is when an OST becomes unavailable, because of a network partition, OSS node crash, etc. When this happens, the OST's clients pause and wait for the OST to become available again, either on the primary OSS or a failover OSS. When the OST comes back online, Lustre starts a recovery process to enable clients to reconnect to the OST. Lustre servers put a limit on the time they will wait in recovery for clients to reconnect[4].

During recovery, clients reconnect and replay their requests, serially, in the same order they were done originally.[5] Periodically, a progress message prints to the log, stating how_many/expected clients have reconnected. If the recovery is aborted, this log shows how many clients managed to reconnect. When all clients have completed recovery, or if the recovery timeout is reached, the recovery period ends and the OST resumes normal request processing.

If some clients fail to replay their requests during the recovery period, this will not stop the recovery from completing. You may have a situation where the OST recovers, but some clients are not able to participate in recovery (e.g. network problems or client failure), so they are evicted and their requests are not replayed. This would result in any operations on the evicted clients failing, including in-progress writes, which would cause cached writes to be lost. This is a normal outcome; the recovery cannot wait indefinitely, or the file system would be hung any time a client failed. The lost transactions are an unfortunate result of the recovery process.

4. The timeout length is determined by the obd_timeout parameter.

5. Until a client receives a confirmation that a given transaction has been written to stable storage, the client holds on to the transaction, in case it needs to be replayed.

Note – Lustre 1.8 introduces the version-based recovery (VBR) feature, which enables a failed client to be ''skipped'', so remaining clients can replay their requests, resulting in a more successful recovery from a downed OST. For more information about the VBR feature, see Version-based Recovery.

In Lustre 1.6 and earlier, the success of the recovery process was limited by uncommitted client requests that are unable to be replayed. Because clients attempted to replay their requests to the OST and MDT in serial order, a client that could not replay its requests causes the recovery stream to stop, and left the remaining clients without an opportunity to reconnect and replay their requests.

22.4.2 Write Performance Better Than Read Performance

Typically, the performance of write operations on a Lustre cluster is better than read operations. When doing writes, all clients are sending write RPCs asynchronously. The RPCs are allocated, and written to disk in the order they arrive. In many cases, this allows the back-end storage to aggregate writes efficiently.

In the case of read operations, the reads from clients may come in a different order and need a lot of seeking to get read from the disk. This noticeably hampers the read throughput.

Currently, there is no readahead on the OSTs themselves, though the clients do readahead. If there are lots of clients doing reads it would not be possible to do any readahead in any case because of memory consumption (consider that even a single RPC (1 MB) readahead for 1000 clients would consume 1 GB of RAM).

For file systems that use socklnd (TCP, Ethernet) as interconnect, there is also additional CPU overhead because the client cannot receive data without copying it from the network buffers. In the write case, the client CAN send data without the additional data copy. This means that the client is more likely to become CPU-bound during reads than writes.

22.4.3 OST Object is Missing or Damaged

If the OSS fails to find an object or finds a damaged object, this message appears:

```
OST object missing or damaged (OST "ost1", object 98148, error -2)
```

If the reported error is -2 (-ENOENT, or "No such file or directory"), then the object is missing. This can occur either because the MDS and OST are out of sync, or because an OST object was corrupted and deleted.

If you have recovered the file system from a disk failure by using e2fsck, then unrecoverable objects may have been deleted or moved to /lost+found on the raw OST partition. Because files on the MDS still reference these objects, attempts to access them produce this error.

If you have recovered a backup of the raw MDS or OST partition, then the restored partition is very likely to be out of sync with the rest of your cluster. No matter which server partition you restored from backup, files on the MDS may reference objects which no longer exist (or did not exist when the backup was taken); accessing those files produces this error.

If neither of those descriptions is applicable to your situation, then it is possible that you have discovered a programming error that allowed the servers to get out of sync. Please report this condition to the Lustre group, and we will investigate.

If the reported error is anything else (such as -5, "I/O error"), it likely indicates a storage failure. The low-level file system returns this error if it is unable to read from the storage device.

Suggested Action

If the reported error is -2, you can consider checking in `/lost+found` on your raw OST device, to see if the missing object is there. However, it is likely that this object is lost forever, and that the file that references the object is now partially or completely lost. Restore this file from backup, or salvage what you can and delete it.

If the reported error is anything else, then you should immediately inspect this server for storage problems.

22.4.4 OSTs Become Read-Only

If the SCSI devices are inaccessible to Lustre at the block device level, then ext3 remounts the device read-only to prevent file system corruption. This is a normal behavior. The status in `/proc/fs/lustre/healthcheck` also shows "not healthy" on the affected nodes.

To determine what caused the "not healthy" condition:

- Examine the consoles of all servers for any error indications
- Examine the syslogs of all servers for any LustreErrors or LBUG
- Check the health of your system hardware and network. (Are the disks working as expected, is the network dropping packets?)
- Consider what was happening on the cluster at the time. Does this relate to a specific user workload or a system load condition? Is the condition reproducible? Does it happen at a specific time (day, week or month)?

To recover from this problem, you must restart Lustre services using these file systems. There is no other way to know that the I/O made it to disk, and the state of the cache may be inconsistent with what is on disk.

22.4.5 Identifying a Missing OST

If an OST is missing for any reason, you may need to know what files are affected. Although an OST is missing, the files system should be operational. From any mounted client node, generate a list of files that reside on the affected OST. It is advisable to mark the missing OST as 'unavailable' so clients and the MDS do not time out trying to contact it.

1. **Generate a list of devices and determine the OST's device number. Run:**

   ```
   $ lctl dl
   ```

   The `lctl dl` command output lists the device name and number, along with the device UUID and the number of references on the device.

2. **Deactivate the OST (on the OSS at the MDS). Run:**

   ```
   $ lctl --device <OST device name or number> deactivate
   ```

   The OST device number or device name is generated by the `lctl dl` command.

   The `deactivate` command prevents clients from creating new objects on the specified OST, although you can still access the OST for reading.

Note – If the OST later becomes available it needs to be reactivated, run:

```
# lctl --device <OST device name or number> activate
```

3. **Determine all the files that are striped over the missing OST, run:**

   ```
   # lfs find -R -o {OST_UUID} /mountpoint
   ```

   This returns a simple list of filenames from the affected file system.

4. **If necessary, you can read the valid parts of a striped file, run:**

   ```
   # dd if=filename of=new_filename bs=4k conv=sync,noerror
   ```

5. **You can delete these files with the unlink or munlink command.**

   ```
   # unlink|munlink filename {filename ...}
   ```

Note – There is no functional difference between the `unlink` and `munlink` commands. The `unlink` command is for newer Linux distributions. You can run `munlink` if `unlink` is not available.

When you run the `unlink` or `munlink` command, the file on the MDS is permanently removed.

6. **If you need to know, specifically, which parts of the file are missing data, then you first need to determine the file layout (striping pattern), which includes the index of the missing OST). Run:**

   ```
   # lfs getstripe -v {filename}
   ```

7. **Use this computation is to determine which offsets in the file are affected: [(C*N + X)*S, (C*N + X)*S + S - 1], N = { 0, 1, 2, ...}**

   where:

   C = stripe count

   S = stripe size

   X = index of bad OST for this file

For example, for a 2 stripe file, stripe size = 1M, the bad OST is at index 0, and you have holes in the file at: [(2*N + 0)*1M, (2*N + 0)*1M + 1M - 1], N = { 0, 1, 2, ...}

If the file system cannot be mounted, currently there is no way that parses metadata directly from an MDS. If the bad OST does not start, options to mount the file system are to provide a loop device OST in its place or replace it with a newly-formatted OST. In that case, the missing objects are created and are read as zero-filled.

In releases prior to Lustre 1.8, you could not mount a file system with a missing OST.

22.4.6 Improving Lustre Performance When Working with Small Files

A Lustre environment where an application writes small file chunks from many clients to a single file will result in bad I/O performance. To improve Lustre's performance with small files:

- Have the application aggregate writes some amount before submitting them to Lustre. By default, Lustre enforces POSIX coherency semantics, so it results in lock ping-pong between client nodes if they are all writing to the same file at one time.
- Have the application do 4kB O_DIRECT sized I/O to the file and disable locking on the output file. This avoids partial-page IO submissions and, by disabling locking, you avoid contention between clients.
- Have the application write contiguous data.
- Add more disks or use SSD disks for the OSTs. This dramatically improves the IOPS rate. Consider creating larger OSTs rather than many smaller OSTs due to less overhead (journal, connections, etc).
- Use RAID-1+0 OSTs instead of RAID-5/6. There is RAID parity overhead for writing small chunks of data to disk.

22.4.7 Default Striping

These are the default striping settings:

```
lov.stripesize=<bytes>

lov.stripecount=<count>

lov.stripeoffset=<offset>
```

To change the default striping information.

- On the MGS:

```
$ lctl conf_param testfs-MDT0000.lov.stripesize=4M
```

- On the MDT and clients:

```
$ mdt/cli> cat /proc/fs/lustre/lov/testfs-{mdt|cli}lov/stripe*
```

22.4.8 Erasing a File System

If you want to erase a file system, run this command on your targets:

```
$ "mkfs.lustre –reformat"
```

If you are using a separate MGS and want to keep other file systems defined on that MGS, then set the `writeconf` flag on the MDT for that file system. The `writeconf` flag causes the configuration logs to be erased; they are regenerated the next time the servers start.

To set the `writeconf` flag on the MDT:

1. **Unmount all clients/servers using this file system, run:**

   ```
   $ umount /mnt/lustre
   ```

2. **Erase the file system and, presumably, replace it with another file system, run:**

   ```
   $ mkfs.lustre –reformat --fsname spfs --mdt --mgs /dev/sda
   ```

3. **If you have a separate MGS (that you do not want to reformat), then add the "writeconf" flag to mkfs.lustre on the MDT, run:**

   ```
   $ mkfs.lustre --reformat --writeconf –fsname spfs --mdt \
   --mgs /dev/sda
   ```

Note – If you have a combined MGS/MDT, reformatting the MDT reformats the MGS as well, causing all configuration information to be lost; you can start building your new file system. Nothing needs to be done with old disks that will not be part of the new file system, just do not mount them.

22.4.9 Reclaiming Reserved Disk Space

All current Lustre installations run the ext3 file system internally on service nodes. By default, the ext3 reserves 5% of the disk space for the root user. In order to reclaim this space, run the following command on your OSSs:

```
tune2fs [-m reserved_blocks_percent] [device]
```

You do not need to shut down Lustre before running this command or restart it afterwards.

22.4.10 Considerations in Connecting a SAN with Lustre

Depending on your cluster size and workload, you may want to connect a SAN with Lustre. Before making this connection, consider the following:

- In many SAN file systems without Lustre, clients allocate and lock blocks or inodes individually as they are updated. The Lustre design avoids the high contention that some of these blocks and inodes may have.
- Lustre is highly scalable and can have a very large number of clients. SAN switches do not scale to a large number of nodes, and the cost per port of a SAN is generally higher than other networking.
- File systems that allow direct-to-SAN access from the clients have a security risk because clients can potentially read any data on the SAN disks, and misbehaving clients can corrupt the file system for many reasons like improper file system, network, or other kernel software, bad cabling, bad memory, and so on. The risk increases with increase in the number of clients directly accessing the storage.

22.4.11 Handling/Debugging "Bind: Address already in use" Error

During startup, Lustre may report a `bind: Address already in use` error and reject to start the operation. This is caused by a portmap service (often NFS locking) which starts before Lustre and binds to the default port 988. You must have port 988 open from firewall or IP tables for incoming connections on the client, OSS, and MDS nodes. LNET will create three outgoing connections on available, reserved ports to each client-server pair, starting with 1023, 1022 and 1021.

Unfortunately, you cannot set sunprc to avoid port 988. If you receive this error, do the following:

- Start Lustre before starting any service that uses sunrpc.
- Use a port other than 988 for Lustre. This is configured in `/etc/modprobe.conf` as an option to the LNET module. For example:

  ```
  options lnet accept_port=988
  ```

- Add `modprobe ptlrpc` to your system startup scripts before the service that uses sunrpc. This causes Lustre to bind to port 988 and sunrpc to select a different port.

Note – You can also use the `sysctl` command to mitigate the NFS client from grabbing the Lustre service port. However, this is a partial workaround as other user-space RPC servers still have the ability to grab the port.

22.4.12 Replacing An Existing OST or MDS

The OST file system is an ldiskfs file system, which is simply a normal ext3 file system plus some performance enhancements—making if very close, in fact, to ext4. To copy the contents of an existing OST to a new OST (or an old MDS to a new MDS), use one of these methods:

- Connect the old OST disk and new OST disk to a single machine, mount both, and use rsync to copy all data between the OST file systems.

  For example:

  ```
  mount -t ldiskfs /dev/old /mnt/ost_old
  mount -t ldiskfs /dev/new /mnt/ost_new
  rsync -aSv /mnt/ost_old/ /mnt/ost_new
  # note trailing slash on ost_old/
  ```

- If you are unable to connect both sets of disk to the same computer, use `rsync` to copy over the network using `rsh` (or `ssh` with `-e ssh`):

  ```
  rsync -aSvz /mnt/ost_old/ new_ost_node:/mnt/ost_new
  ```

- Use the same procedure for the MDS, with one additional step:

  ```
  cd /mnt/mds_old; getfattr -R -e base64 -d . > /tmp/mdsea; \
  <copy all MDS files as above>; cd /mnt/mds_new; setfattr \
  --restore=/tmp/mdsea
  ```

22.4.13 Handling/Debugging Error "- 28"

Linux error -28 is -ENOSPC and indicates that the file system has run out of space. You need to create larger file systems for the OSTs. Normally, Lustre reports this to your application. If the application is checking the return code from its function calls, then it decodes it into a textual error message like "No space left on device." It also appears in the system log messages.

During a "write" or "sync" operation, the file in question resides on an OST which is already full. New files that are created do not use full OSTs, but existing files continue to use the same OST. You need to expand the specific OST or copy/stripe the file over to an OST with more space available. You encounter this situation occasionally when creating files, which may indicate that your MDS has run out of inodes and needs to be enlarged. To check this, use `df -i`

You may also receive this error if the MDS runs out of free blocks. Since the output of `df` is an aggregate of the data from the MDS and all of the OSTs, it may not show that the file system is full when one of the OSTs has run out of space. To determine which OST or MDS is running out of space, check the free space and inodes on a client:

```
grep '[0-9]' /proc/fs/lustre/osc/*/kbytes{free,avail,total}
grep '[0-9]' /proc/fs/lustre/osc/*/files{free,total}
grep '[0-9]' /proc/fs/lustre/mdc/*/kbytes{free,avail,total}
grep '[0-9]' /proc/fs/lustre/mdc/*/files{free,total}
```

You can find other numeric error codes in /usr/include/asm/errno.h along with their short name and text description.

22.4.14 Triggering Watchdog for PID NNN

In some cases, a server node triggers a watchdog timer and this causes a process stack to be dumped to the console along with a Lustre kernel debug log being dumped into `/tmp` (by default). The presence of a watchdog timer does NOT mean that the thread OOPSed, but rather that it is taking longer time than expected to complete a given operation. In some cases, this situation is expected.

For example, if a RAID rebuild is really slowing down I/O on an OST, it might trigger watchdog timers to trip. But another message follows shortly thereafter, indicating that the thread in question has completed processing (after some number of seconds). Generally, this indicates a transient problem. In other cases, it may legitimately signal that a thread is stuck because of a software error (lock inversion, for example).

```
Lustre: 0:0:(watchdog.c:122:lcw_cb())
```

The above message indicates that the watchdog is active for pid 933:

It was inactive for 100000ms:

```
Lustre: 0:0:(linux-debug.c:132:portals_debug_dumpstack())
```

Showing stack for process:

```
933 ll_ost_25     D F896071A     0   933      1    934   932 (L-TLB)
f6d87c60 00000046 00000000 f896071a f8def7cc 00002710 00001822 2da48cae
0008cf1a f6d7c220 f6d7c3d0 f6d86000 f3529648 f6d87cc4 f3529640 f8961d3d
00000010 f6d87c9c ca65a13c 00001fff 00000001 00000001 00000000 00000001
```

Call trace:

```
filter_do_bio+0x3dd/0xb90 [obdfilter]
default_wake_function+0x0/0x20
filter_direct_io+0x2fb/0x990 [obdfilter]
filter_preprw_read+0x5c5/0xe00 [obdfilter]
lustre_swab_niobuf_remote+0x0/0x30 [ptlrpc]
ost_brw_read+0x18df/0x2400 [ost]
ost_handle+0x14c2/0x42d0 [ost]
ptlrpc_server_handle_request+0x870/0x10b0 [ptlrpc]
ptlrpc_main+0x42e/0x7c0 [ptlrpc]
```

22.4.15 Handling Timeouts on Initial Lustre Setup

If you come across timeouts or hangs on the initial setup of your Lustre system, verify that name resolution for servers and clients is working correctly. Some distributions configure `/etc/hosts sts` so the name of the local machine (as reported by the 'hostname' command) is mapped to local host (127.0.0.1) instead of a proper IP address.

This might produce this error:

```
LustreError:(ldlm_handle_cancel()) received cancel for unknown lock cookie
0xe74021a4b41b954e from nid 0x7f000001 (0:127.0.0.1)
```

22.4.16 Handling/Debugging "LustreError: xxx went back in time"

Each time Lustre changes the state of the disk file system, it records a unique transaction number. Occasionally, when committing these transactions to the disk, the last committed transaction number displays to other nodes in the cluster to assist the recovery. Therefore, the promised transactions remain absolutely safe on the disappeared disk.

This situation arises when:

- You are using a disk device that claims to have data written to disk before it actually does, as in case of a device with a large cache. If that disk device crashes or loses power in a way that causes the loss of the cache, there can be a loss of transactions that you believe are committed. This is a very serious event, and you should run e2fsck against that storage before restarting Lustre.
- As per the Lustre requirement, the shared storage used for failover is completely cache-coherent. This ensures that if one server takes over for another, it sees the most up-to-date and accurate copy of the data. In case of the failover of the server, if the shared storage does not provide cache coherency between all of its ports, then Lustre can produce an error.

If you know the exact reason for the error, then it is safe to proceed with no further action. If you do not know the reason, then this is a serious issue and you should explore it with your disk vendor.

If the error occurs during failover, examine your disk cache settings. If it occurs after a restart without failover, try to determine how the disk can report that a write succeeded, then lose the Data Device corruption or Disk Errors.

22.4.17 Lustre Error: "Slow Start_Page_Write"

The `slow start_page_write` message appears when the operation takes an extremely long time to allocate a batch of memory pages. Use these pages to receive network traffic first, and then write to disk.

22.4.18 Drawbacks in Doing Multi-client O_APPEND Writes

It is possible to do multi-client O_APPEND writes to a single file, but there are few drawbacks that may make this a sub-optimal solution. These drawbacks are:

- Each client needs to take an EOF lock on all the OSTs, as it is difficult to know which OST holds the end of the file until you check all the OSTs. As all the clients are using the same O_APPEND, there is significant locking overhead.
- The second client cannot get all locks until the end of the writing of the first client, as the taking serializes all writes from the clients.
- To avoid deadlocks, the taking of these locks occurs in a known, consistent order. As a client cannot know which OST holds the next piece of the file until the client has locks on all OSTS, there is a need of these locks in case of a striped file.

22.4.19 Slowdown Occurs During Lustre Startup

When Lustre starts, the Lustre file system needs to read in data from the disk. For the very first mdsrate run after the reboot, the MDS needs to wait on all the OSTs for object pre-creation. This causes a slowdown to occur when Lustre starts up.

After the file system has been running for some time, it contains more data in cache and hence, the variability caused by reading critical metadata from disk is mostly eliminated. The file system now reads data from the cache.

22.4.20 Log Message ‘Out of Memory’ on OST

When planning the hardware for an OSS node, consider the memory usage of several components in the Lustre system. If insufficient memory is available, an ‘out of memory’ message can be logged.

During normal operation, several conditions indicate insufficient RAM on a server node:

- kernel "Out of memory" and/or "oom-killer" messages
- Lustre "kmalloc of 'mmm' (NNNN bytes) failed..." messages
- Lustre or kernel stack traces showing processes stuck in "try_to_free_pages"

For information on determining the MDS memory and OSS memory requirements, see Memory Requirements.

22.4.21 Number of OSTs Needed for Sustained Throughput

The number of OSTs required for sustained throughput depends on your hardware configuration. If you are adding an OST that is identical to an existing OST, you can use the speed of the existing OST to determine how many more OSTs to add.

Keep in mind that adding OSTs affects resource limitations, such as bus bandwidth in the OSS and network bandwidth of the OSS interconnect. You need to understand the performance capability of all system components to develop an overall design that meets your performance goals and scales to future system requirements.

Note – For best performance, put the MGS and MDT on separate devices.

22.4.22 Setting SCSI I/O Sizes

Some SCSI drivers default to a maximum I/O size that is too small for good Lustre performance. we have fixed quite a few drivers, but you may still find that some drivers give unsatisfactory performance with Lustre. As the default value is hard-coded, you need to recompile the drivers to change their default. On the other hand, some drivers may have a wrong default set.

If you suspect bad I/O performance and an analysis of Lustre statistics indicates that I/O is not 1 MB, check `/sys/block/<device>/queue/max_sectors_kb`. If the `max_sectors_kb` value is less than 1024, set it to at least 1024 to improve performance. If changing `max_sectors_kb` does not change the I/O size as reported by Lustre, you may want to examine the SCSI driver code.

22.4.23 Identifying Which Lustre File an OST Object Belongs To

Use this procedure to identify the file containing a given object on a given OST.

1. **On the OST (as root), run debugfs to display the FID[6] of the file associated with the object.**

   For example, if the object is 34976 on /dev/lustre/ost_test2, the debug command is:

   ```
   # debugfs -c -R "stat /O/0/d$((34976 %32))/34976" /dev/lustre/ost_test2
   ```

   The command output is:

   ```
   debugfs 1.41.5.sun2 (23-Apr-2009)
   /dev/lustre/ost_test2: catastrophic mode - not reading inode or
   group bitmaps
   Inode: 352365   Type: regular    Mode:  0666   Flags: 0x80000
   Generation: 1574463214    Version: 0xea020000:00000000
   User:   500   Group:   500   Size: 260096
   File ACL: 0    Directory ACL: 0
   Links: 1   Blockcount: 512
   Fragment:  Address: 0    Number: 0    Size: 0
   ctime: 0x4a216b48:00000000 -- Sat May 30 13:22:16 2009
   atime: 0x4a216b48:00000000 -- Sat May 30 13:22:16 2009
   mtime: 0x4a216b48:00000000 -- Sat May 30 13:22:16 2009
   crtime: 0x4a216b3c:975870dc -- Sat May 30 13:22:04 2009
   Size of extra inode fields: 24
   Extended attributes stored in inode body:
   fid = "e2 00 11 00 00 00 00 00 25 43 c1 87 00 00 00 00 a0 88 00 00
   00 00 00 00 00 00 00 00 00 00 00 00 " (32)
   BLOCKS:
   (0-63):47968-48031
   TOTAL: 64
   ```

6. The FID is the file identifier.

2. **Note the FID's EA and apply it to the osd_inode_id mapping.**

   In this example, the FID's EA is:

   ```
   e20011000000000002543c18700000000a088000000000000000000000000000000
   ```

   ```
   struct osd_inode_id {

     __u64 oii_ino; /* inode number */
     __u32 oii_gen; /* inode generation */
     __u32 oii_pad; /* alignment padding */
   };
   ```

   After swapping, you get an inode number of 0x001100e2 and generation of 0.

3. **On the MDT (as root), use debugfs to find the file associated with the inode.**

   ```
   # debugfs -c -R "ncheck 0x001100e2" /dev/lustre/mdt_test
   ```

   Here is the command output:

   ```
   debugfs 1.41.5.sun2 (23-Apr-2009)
   /dev/lustre/mdt_test: catastrophic mode - not reading inode or group bitmaps
   Inode    Pathname
   1114338  /ROOT/brian-laptop-guest/clients/client11/~dmtmp/PWRPNT/ZD16.BMP
   ```

   The command lists the inode and pathname associated with the object.

Note – Debugfs' "ncheck" is a brute-force search that may take a long time to complete.

Note – To find the Lustre file from a disk LBA, follow the steps listed in the document at this URL: http://smartmontools.sourceforge.net/badblockhowto.html. Then, follow the steps above to resolve the Lustre filename.

CHAPTER 23

Lustre Debugging

This chapter describes tips and information to debug Lustre, and includes the following sections:

Lustre is a complex system that requires a rich debugging environment to help locate problems.

23.1 Lustre Debug Messages

Each Lustre debug message has the tag of the subsystem it originated in, the message type, and the location in the source code. The subsystems and debug types used in Lustre are as follows:

- Standard Subsystems:

  mdc, mds, osc, ost, obdclass, obdfilter, llite, ptlrpc, portals, lnd, ldlm, lov

- Debug Types:

Types	Description
trace	Entry/Exit markers
dlmtrace	Locking-related information
inode	
super	
ext2	Anything from the ext2_debug
malloc	Print malloc or free information
cache	Cache-related information
info	General information
ioctl	IOCTL-related information
blocks	Ext2 block allocation information
net	Networking
warning	
buffs	
other	
dentry	
portals	Entry/Exit markers
page	Bulk page handling
error	Error messages
emerg	
rpctrace	For distributed debugging
ha	Failover and recovery-related information

23.1.1 Format of Lustre Debug Messages

Lustre uses the CDEBUG and CERROR macros to print the debug or error messages. To print the message, the CDEBUG macro uses `portals_debug_msg` (`portals/linux/oslib/debug.c`). The message format is described below, along with an example.

Parameter	Description
subsystem	800000
debug mask	000010
smp_processor_id	0
sec.used	10818808 47.677302
stack size	1204:
pid	2973:
host pid (if uml) or zero	31070:
(file:line #:functional())	(as_dev.c:144:create_write_buffers())
debug message	kmalloced '*obj': 24 at a375571c (tot 17447717)

23.2 Tools for Lustre Debugging

The Lustre system offers debugging tools combined by the operating system and Lustre itself. These tools are:

- **Debug logs**: A circular debug buffer holds a substantial amount of debugging information (MBs or more) during the first insertion of the kernel module. When this buffer fills up, it wraps and discards the oldest information. Lustre offers additional debug messages that can be written out to this kernel log.

  The debug log holds Lustre internal logging, separate from the error messages printed to syslog or console. Entries to the Lustre debug log are controlled by the mask set by `/proc/sys/lnet/debug`. The log defaults to 5 MB per CPU, and is a ring buffer. Newer messages overwrite older ones. The default log size can be increased, as a busy system will quickly overwrite the 5 MB default.
- **Debug daemon**: The debug daemon controls logging of debug messages.
- **/proc/sys/lnet/debug**: This log contains a mask that can be used to delimit the debugging information written out to the kernel debug logs.
- **lctl**: This tool is used to manually dump the log and post-process logs that are dumped automatically.
- **leak_finder.pl**: This is useful program which helps find memory leaks in the code.
- **strace**: This tool allows a system call to be traced.
- **/var/log/messages**: syslogd prints fatal or serious messages at this log.
- **Crash dumps**: On crash-dump enabled kernels, `sysrq c` produces a crash dump. Lustre enhances this crash dump with a log dump (the last 64 KB of the log) to the console.
- **debugfs**: Interactive file system debugger.
- **Lustre subsystem asserts**: In case of asserts, a log writes at `/tmp/lustre_log.<timestamp>`.
- **lfs**: This Lustre utility helps get to the extended attributes of a Lustre file (among other things).
- **Lustre diagnostic tool**: This utility helps users report and create logs for Lustre bugs.
- **GNU tar (gtar)**: This modified version of the gtar utility can back up and restore extended attributes (i.e. file striping) for Lustre. Files backed up using gtar are restored per the backed up striping information. The backup procedure does not use default striping rules.

Note – Normal gtar does not store/restore Lustre attributes. To use this functionality, you must download the Lustre-patched tar utility (modified gtar), available here:

http://downloads.lustre.org/public/tools/lustre-tar/

23.2.1 Debug Daemon Option to lctl

The `debug_daemon` allows users to control the Lustre kernel debug daemon to dump the `debug_kernel` buffer to a user-specified file. This functionality uses a kernel thread on top of `debug_kernel`. `debug_kernel`, another sub-command of `lctl`, continues to work in parallel with `debug_daemon` command.

`Debug_daemon` is highly dependent on file system write speed. File system writes operation may not be fast enough to flush out all the `debug_buffer` if Lustre file system is under heavy system load and continue to CDEBUG to the `debug_buffer`. `Debug_daemon` put 'DEBUG MARKER: Trace buffer full' into the `debug_buffer` to indicate debug_buffer is overlapping itself before `debug_daemon` flush data to a file.

Users can use `lctl control` to start or stop Lustre daemon from dumping the `debug_buffer` to a file. Users can also temporarily hold daemon from dumping the file. Use of the `debug_daemon` sub-command to `lctl` can provide the same function.

23.2.1.1 lctl Debug Daemon Commands

This section describes `lctl` daemon debug commands.

$ lctl debug_daemon start [{file} {megabytes}]

Initiates the `debug_daemon` to start dumping debug_buffer into a file. The file can be a system default file, as shown in `/proc/sys/lnet/debug_path`. After Lustre starts, the default path is `/tmp/lustre-log-$HOSTNAME`. Users can specify a new filename for `debug_daemon` to output `debug_buffer`. The new file name shows up in `/proc/sys/lnet/debug_path`. Megabytes is the limitation of the file size in MBs. The daemon wraps around and dumps data to the beginning of the file when the output file size is over the limit of the user-specified file size. To decode the dumped file to ASCII and order the log entries by time, run:

```
lctl debug_file {file} > {newfile}
```

The output is internally sorted by the `lctl` command using quicksort.

debug_daemon stop

Completely shuts down the `debug_daemon` operation and flushes the file output. Otherwise, `debug_daemon` is shut down as part of Lustre file system shutdown process. Users can restart `debug_daemon` by using start command after each stop command issued.

This is an example using `debug_daemon` with the interactive mode of `lctl` to dump debug logs to a 10 MB file.

```
#~/utils/lctl
```

To start daemon to dump `debug_buffer` into a 40 MB /tmp/dump file.

```
lctl > debug_daemon start /trace/log 40
```

To completely shut down the daemon.

```
lctl > debug_daemon stop
```

To start another daemon with an unlimited file size.

```
lctl > debug_daemon start /tmp/unlimited
```

The text message `*** End of debug_daemon trace log ***` appears at the end of each output file.

23.2.2 Controlling the Kernel Debug Log

The amount of information printed to the kernel debug logs can be controlled by masks in `/proc/sys/lnet/subsystem_debug` and `/proc/sys/lnet/debug`. The `subsystem_debug` mask controls subsystems (e.g., obdfilter, net, portals, OSC, etc.) and the debug mask controls debug types written to the log (e.g., info, error, trace, alloc, etc.).

To turn off Lustre debugging completely:

```
sysctl -w lnet.debug=0
```

To turn on full Lustre debugging:

```
sysctl -w lnet.debug=-1
```

To turn on logging of messages related to network communications:

```
sysctl -w lnet.debug=net
```

To turn on logging of messages related to network communications and existing debug flags:

```
sysctl -w lnet.debug=+net
```

To turn off network logging with changing existing flags:

```
sysctl -w lnet.debug=-net
```

The various options available to print to kernel debug logs are listed in `lnet/include/libcfs/libcfs.h`

23.2.3 The lctl Tool

Lustre's source code includes debug messages which are very useful for troubleshooting. As described above, debug messages are subdivided into a number of subsystems and types. This subdivision allows messages to be filtered, so that only messages of interest to the user are displayed. The `lctl` tool is useful to enable this filtering and manipulate the logs to extract the useful information from it. Use `lctl` to obtain the necessary debug messages:

1. **To obtain a list of all the types and subsystems:**

   ```
   lctl > debug_list <subs | types>
   ```

2. **To filter the debug log:**

   ```
   lctl > filter <subsystem name | debug type>
   ```

Note – When `lctl` filters, it removes unwanted lines from the displayed output. This does not affect the contents of the debug log in the kernel's memory. As a result, you can print the log many times with different filtering levels without worrying about losing data.

3. **To show debug messages belonging to certain subsystem or type:**

   ```
   lctl > show <subsystem name | debug type>
   ```

   `debug_kernel` pulls the data from the kernel logs, filters it appropriately, and displays or saves it as per the specified options

   ```
   lctl > debug_kernel [output filename]
   ```

   If the debugging is being done on User Mode Linux (UML), it might be useful to save the logs on the host machine so that they can be used at a later time.

4. **If you already have a debug log saved to disk (likely from a crash), to filter a log on disk:**

   ```
   lctl > debug_file <input filename> [output filename]
   ```

   During the debug session, you can add markers or breaks to the log for any reason:

   ```
   lctl > mark [marker text]
   ```

   The marker text defaults to the current date and time in the debug log (similar to the example shown below):

   ```
   DEBUG MARKER: Tue Mar 5 16:06:44 EST 2002
   ```

5. **To completely flush the kernel debug buffer:**

   ```
   lctl > clear
   ```

Note – Debug messages displayed with `lctl` are also subject to the kernel debug masks; the filters are additive.

23.2.4 Finding Memory Leaks

Memory leaks can occur in a code where you allocate a memory, but forget to free it when it becomes non-essential. You can use the `leak_finder.pl` tool to find memory leaks. Before running this program, you must turn on the debugging to collect all malloc and free entries. Run:

```
sysctl -w lnet.debug=+malloc
```

Dump the log into a user-specified log file using lctl (as shown in The lctl Tool). Run the leak finder on the newly-created log dump:

```
perl leak_finder.pl <ascii-logname>
```

The output is:

```
malloced 8bytes at a3116744 (called pathcopy)
(lprocfs_status.c:lprocfs_add_vars:80)
freed 8bytes at a3116744 (called pathcopy)
(lprocfs_status.c:lprocfs_add_vars:80)
```

The tool displays the following output to show the leaks found:

```
Leak:32bytes allocated at a23a8fc
(service.c:ptlrpc_init_svc:144,debug file line 241)
```

23.2.5 Printing to /var/log/messages

To dump debug messages to the console, set the corresponding debug mask in the `printk` flag:

```
sysctl -w lnet.printk=-1
```

This slows down the system dramatically. It is also possible to selectively enable or disable this for particular flags using:

```
sysctl -w lnet.printk=+vfstrace
sysctl -w lnet.printk=-vfstrace
```

23.2.6 Tracing Lock Traffic

Lustre has a specific debug type category for tracing lock traffic. Use:

```
lctl> filter all_types
lctl> show dlmtrace
lctl> debug_kernel [filename]
```

23.2.7 Sample lctl Run

bash-2.04# ./lctl

```
lctl > debug_kernel /tmp/lustre_logs/log_all
```

Debug log: 324 lines, 324 kept, 0 dropped.

```
lctl > filter trace
```

Disabling output of type "trace"

```
lctl > debug_kernel /tmp/lustre_logs/log_notrace
```

Debug log: 324 lines, 282 kept, 42 dropped.

```
lctl > show trace
```

Enabling output of type "trace"

```
lctl > filter portals
```

Disabling output from subsystem "portals"

```
lctl > debug_kernel /tmp/lustre_logs/log_noportals
```

Debug log: 324 lines, 258 kept, 66 dropped.

23.2.8 Adding Debugging to the Lustre Source Code

In the Lustre source code, the debug infrastructure provides a number of macros which aid in debugging or reporting serious errors. All of these macros depend on having the DEBUG_SUBSYSTEM variable set at the top of the file:

#define DEBUG_SUBSYSTEM S_PORTALS

Macro	Description
LBUG	A panic-style assertion in the kernel which causes Lustre to dump its circular log to the /tmp/lustre-log file. This file can be retrieved after a reboot. LBUG freezes the thread to allow capture of the panic stack. A system reboot is needed to clear the thread.
LASSERT	Validates a given expression as true, otherwise calls LBUG. The failed expression is printed on the console, although the values that make up the expression are not printed.
LASSERTF	Similar to LASSERT but allows a free-format message to be printed, like printf/printk.

Macro	Description
CDEBUG	The basic, most commonly used debug macro that takes just one more argument than standard printf - the debug type. This message adds to the debug log with the debug mask set accordingly. Later, when a user retrieves the log for troubleshooting, they can filter based on this type. CDEBUG(D_INFO, "This is my debug message: the number is %d\n", number).
CERROR	Behaves similarly to CDEBUG, but unconditionally prints the message in the debug log and to the console. This is appropriate for serious errors or fatal conditions: CERROR("Something very bad has happened, and the return code is %d.\n", rc);
ENTRY and EXIT	Add messages to aid in call tracing (takes no arguments). When using these macros, cover all exit conditions to avoid confusion when the debug log reports that a function was entered, but never exited.
LDLM_DEBUG and LDLM_DEBUG_NOLOCK	Used when tracing MDS and VFS operations for locking. These macros build a thin trace that shows the protocol exchanges between nodes.
DEBUG_REQ	Prints information about the given ptlrpc_request structure.
OBD_FAIL_CHECK	Allows insertion of failure points into the Lustre code. This is useful to generate regression tests that can hit a very specific sequence of events. This works in conjunction with "sysctl -w lustre.fail_loc={fail_loc}" to set a specific failure point for which a given OBD_FAIL_CHECK will test.
OBD_FAIL_TIMEOUT	Similar to OBD_FAIL_CHECK. Useful to simulate hung, blocked or busy processes or network devices. If the given fail_loc is hit, OBD_FAIL_TIMEOUT waits for the specified number of seconds.
OBD_RACE	Similar to OBD_FAIL_CHECK. Useful to have multiple processes execute the same code concurrently to provoke locking races. The first process to hit OBD_RACE sleeps until a second process hits OBD_RACE, then both processes continue.
OBD_FAIL_ONCE	A flag set on a lustre.fail_loc breakpoint to cause the OBD_FAIL_CHECK condition to be hit only one time. Otherwise, a fail_loc is permanent until it is cleared with "sysctl -w lustre.fail_loc=0".

Macro	Description
OBD_FAIL_RAND	Has OBD_FAIL_CHECK fail randomly; on average every (1 / lustre.fail_val) times.
OBD_FAIL_SKIP	Has OBD_FAIL_CHECK succeed lustre.fail_val times, and then fail permanently or once with OBD_FAIL_ONCE.
OBD_FAIL_SOME	Has OBD_FAIL_CHECK fail lustre.fail_val times, and then succeed.

23.3 Troubleshooting with strace

The operating system makes `strace` (program trace utility) available. Use `strace` to trace program execution. The `strace` utility pauses programs made by a process and records the system call, arguments, and return values. This is a very useful tool, especially when you try to troubleshoot a failed system call.

To invoke `strace` on a program:

```
$ strace <program> <args>
```

Sometimes, a system call may fork child processes. In this situation, use the `-f` option of `strace` to trace the child processes:

```
$ strace -f <program> <args>
```

To redirect the `strace` output to a file (to review at a later time):

```
$ strace -o <filename> <program> <args>
```

Use the `-ff` option, along with `-o`, to save the trace output in `filename.pid`, where pid is the process ID of the process being traced. Use the `-ttt` option to timestamp all lines in the `strace` output, so they can be correlated to operations in the lustre kernel debug log.

If the debugging is done in UML, save the traces on the host machine. In this example, hostfs is mounted on `/r`:

```
$ strace -o /r/tmp/vi.strace
```

23.4 Looking at Disk Content

In Lustre, the inodes on the metadata server contain extended attributes (EAs) that store information about file striping. EAs contain a list of all object IDs and their locations (that is, the OST that stores them). The lfs tool can be used to obtain this information for a given file via the `getstripe` sub-command. Use a corresponding lfs setstripe command to specify striping attributes for a new file or directory.

The `lfs getstripe` utility is written in C; it takes a Lustre filename as input and lists all the objects that form a part of this file. To obtain this information for the file /mnt/lustre/frog in Lustre file system, run:

```
$ lfs getstripe /mnt/lustre/frog
$
   OBDs:
     0 : OSC_localhost_UUID
     1: OSC_localhost_2_UUID
     2: OSC_localhost_3_UUID
   obdix         objid
   0             17
   1             4
```

The debugfs tool is provided by the e2fsprogs package. It can be used for interactive debugging of an ext3/ldiskfs file system. The debugfs tool can either be used to check status or modify information in the file system. In Lustre, all objects that belong to a file are stored in an underlying ldiskfs file system on the OST's. The file system uses the object IDs as the file names. Once the object IDs are known, use the debugfs tool to obtain the attributes of all objects from different OST's. A sample run for the /mnt/lustre/frog file used in the above example is shown here:

```
   $ debugfs -c /tmp/ost1
   debugfs: cd O
   debugfs: cd 0                  /* for files in group 0 */
   debugfs: cd d<objid % 32>
   debugfs: stat <objid>          /* for getattr on object */
   debugfs: quit
## Suppose object id is 36, then follow the steps below:
   $ debugfs /tmp/ost1
   debugfs: cd O
   debugfs: cd 0
   debugfs: cd d4                 /* objid % 32 */
   debugfs: stat 36               /* for getattr on obj 4*/
   debugfs: dump 36 /tmp/obj.36   /* dump contents of obj 4 */
   debugfs: quit
```

23.4.1 Determine the Lustre UUID of an OST

To determine the Lustre UUID of an obdfilter disk (for example, if you mix up the cables on your OST devices or the SCSI bus numbering suddenly changes and the SCSI devices get new names), use debugfs to get the last_rcvd file.

23.4.2 Tcpdump

Lustre provides a modified version of tcpdump which helps to decode the complete Lustre message packet. This tool has more support to read packets from clients to OSTs, than to decode packets between clients and MDSs. The tcpdump module is available from Lustre CVS at www.sourceforge.net

It can be checked out as:

```
cvs co -d :ext:<username>@cvs.lustre.org:/cvsroot/lustre tcpdump
```

23.5 Ptlrpc Request History

Each service always maintains request history, which is useful for first occurrence troubleshooting. Ptlrpc history works as follows:

1. **Request_in_callback() adds the new request to the service's request history.**
2. **When a request buffer becomes idle, add it to the service's request buffer history list.**
3. **Cull buffers from the service's request buffer history if it has grown above** "req_buffer_history_max" and remove its reqs from the service's request history.

Request history is accessed/controlled via the following `/proc` files under the service directory.

- `req_buffer_history_len`

  Number of request buffers currently in the history
- `req_buffer_history_max`

  Maximum number of request buffers to keep
- `req_history`

  The request history

Requests in the history include "live" requests that are actually being handled. Each line in "req_history" looks like:

```
<seq>:<target NID>:<client ID>:<xid>:<length>:<phase> <svc specific>
```

Parameter	Description
seq	Request sequence number
target NID	Destination NID of the incoming request
client ID	Client PID and NID
xid	rq_xid
length	Size of the request message
phase	• New (waiting to be handled or could not be unpacked) • Interpret (unpacked or being handled) • Complete (handled)
svc specific	Service-specific request printout. Currently, the only service that does this is the OST (which prints the opcode if the message has been unpacked successfully

23.6 Using LWT Tracing

Lustre offers a very lightweight tracing facility called LWT. It prints fixed size requests into a buffer and is much faster than LDEBUG. The LWT tracking facility is very successful to debug difficult problems.

LWT trace-based records that are dumped contain:

- Current CPU
- Process counter
- Pointer to file
- Pointer to line in the file
- 4 void * pointers

An `lctl` command dumps the logs to files.

PART IV Lustre for Users

This part includes chapters on Lustre striping and I/O options, security and operating tips.

CHAPTER 24

Striping and I/O Options

This chapter describes file striping and I/O options, and includes the following sections:

24.1 File Striping

Lustre stores files of one or more objects on OSTs. When a file is comprised of more than one object, Lustre stripes the file data across them in a round-robin fashion. Users can configure the number of stripes, the size of each stripe, and the servers that are used.

One of the most frequently-asked Lustre questions is "*How should I stripe my files, and what is a good default?*" The short answer is that it depends on your needs. A good rule of thumb is to stripe over as few objects as will meet those needs and no more.

24.1.1 Advantages of Striping

There are two reasons to create files of multiple stripes: bandwidth and size.

24.1.1.1 Bandwidth

There are many applications which require high-bandwidth access to a single file – more bandwidth than can be provided by a single OSS. For example, scientific applications which write to a single file from hundreds of nodes or a binary executable which is loaded by many nodes when an application starts.

In cases like these, stripe your file over as many OSSs as it takes to achieve the required peak aggregate bandwidth for that file. This strategy is known as 'large striping', the ability to stripe across a larger number of OSSs. Large striping should only be used when the file size is very large and/or is accessed by many nodes at a time. Currently, Lustre files can be striped across up to 160 OSSs, the maximum stripe count for an ext3 file system.

Large striping can improve performance if the aggregate client bandwidth exceeds the server bandwidth, and the application reads/writes data fast enough to take advantage of the additional OSS bandwidth. The largest useful stripe count is bounded by the I/O rate of your clients/jobs divided by the performance per OSS.

The second reason to stripe is when a single OST does not have enough free space to hold the entire file.

There is never an exact, one-to-one mapping between clients and OSTs. Lustre uses a round-robin algorithm for OST stripe selection until free space on OSTs differ by more than 20%. However, depending on actual file sizes, some stripes may be mostly empty, while others are more full. For a more detailed description of stripe assignments, see Managing Free Space.

After every ostcount+1 objects, Lustre skips an OST. This causes Lustre's "starting point" to precess around, eliminating some degenerated cases where applications that create very regular file layouts (striping patterns) would have preferentially used a particular OST in the sequence.

24.1.2 Disadvantages of Striping

There are two disadvantages to striping which should deter you from choosing a default policy that stripes over all OSTs unless you really need it: increased overhead and increased risk.

24.1.2.1 Increased Overhead

Increased overhead comes in the form of extra network operations during common operations such as stat and unlink, and more locks. Even when these operations are performed in parallel, there is a big difference between doing 1 network operation and 100 operations.

Increased overhead also comes in the form of server contention. Consider a cluster with 100 clients and 100 OSSs, each with one OST. If each file has exactly one object and the load is distributed evenly, there is no contention and the disks on each server can manage sequential I/O. If each file has 100 objects, then the clients all compete with one another for the attention of the servers, and the disks on each node seek in 100 different directions. In this case, there is needless contention.

24.1.2.2 Increased Risk

Increased risk is evident when you consider the example of striping each file across all servers. In this case, if any one OSS catches on-fire, a small part of every file is lost. By comparison, if each file has exactly one stripe, you lose fewer files, but you lose them in their entirety. Most users would rather lose some of their files entirely than all of their files partially.

24.1.3 Stripe Size

Choosing a stripe size is a small balancing act, but there are reasonable defaults. The stripe size must be a multiple of the page size. For safety, Lustre's tools enforce a multiple of 64 KB (the maximum page size on ia64 and PPC64 nodes), so users on platforms with smaller pages do not accidentally create files which might cause problems for ia64 clients.

Although you can create files with a stripe size of 64 KB, this is a poor choice. Practically, the smallest recommended stripe size is 512 KB because Lustre sends 1 MB chunks over the network. This is a good amount of data to transfer at one time. Choosing a smaller stripe size may hinder the batching.

Generally, a good stripe size for sequential I/O using high-speed networks is between 1 MB and 4 MB. In most situations, stripe sizes larger than 4 MB do not parallelize as effectively because Lustre tries to keep the amount of dirty cached data below 32 MB per server (with the default configuration).

In an upcoming release, the 'wide striping' feature will be introduced, supporting stripe sizes up to 4 GB. Wide striping can be used to improve performance with very large files although, depending on the configuration, it can be counterproductive after a certain stripe size.

Writes which cross an object boundary are slightly less efficient than writes which go entirely to one server. Depending on your application's write patterns, you can assist it by choosing a stripe size with that in mind. If the file is written in a very consistent and aligned way, make the stripe size a multiple of the write() size.

The choice of stripe size has no effect on a single-stripe file.

24.2 Displaying Files and Directories with lfs getstripe

Use `lfs` to print the index and UUID for each OST in the file system, along with the OST index and object ID for each stripe in the file. For directories, the default settings for files created in that directory are printed.

```
lfs getstripe <filename>
```

Use `lfs find` to inspect an entire tree of files.

```
lfs find [--recursive | -r] <file or directory> ...
```

If a process creates a file, use the `lfs getstripe` command to determine which OST(s) the file resides on.

Using 'cat' as an example, run:

```
$ cat > foo
```

In another terminal, run:

```
$ lfs getstripe /barn/users/jacob/tmp/foo
OBDS
```

You can also use `ls -l /proc/<pid>/fd/` to find open files using Lustre, run:

```
$ lfs getstripe $(readlink /proc/$(pidof cat)/fd/1)
```

```
OBDS:
0: databarn-ost1_UUID ACTIVE
1: databarn-ost2_UUID ACTIVE
2: databarn-ost3_UUID ACTIVE
3: databarn-ost4_UUID ACTIVE
/barn/users/jacob/tmp/foo
          obdidx    objid         objid         group
          2         835487        0xcbf9f       0
```

This shows that the file lives on obdidx 2, which is databarn-ost3. To see which node is serving that OST, run:

```
$ cat /proc/fs/lustre/osc/*databarn-ost3*/ost_conn_uuid
NID_oss1.databarn.87k.net_UUID
```

The above condition/operation also works with connections to the MDS. For that, replace `osc` with `mdc` and `ost` with `mds` in the above commands.

24.3 lfs setstripe – Setting File Layouts

Use the `lfs setstripe` command to create new files with a specific file layout (stripe pattern) configuration.

```
lfs setstripe [--size|-s stripe-size] [--count|-c stripe-cnt]
[--index|-i start-ost] <filename|dirname>
```

stripe-size

Stripe size is how much data to write to one OST before moving to the next OST. The default stripe-size is 1 MB, and passing a stripe-size of 0 causes the default stripe size to be used. Otherwise, the stripe-size must be a multiple of 64 KB.

stripe-count

Stripe count is how many OSTs to use. The default stripe-count is 1, and passing a stripe-count of 0 causes the default stripe count to be used. A stripe-count of -1 means always stripe over all OSTs.

start-ost

Start ost is the first OST to which files are written. The default start-ost is -1, and passing a start-ost of -1 allows the MDS to choose the starting index. This setting is strongly recommended, as it allows space and load balancing to be done by the MDS as needed. Otherwise, the file starts on the specified OST index, starting at zero (0).

Note – If you pass a start-ost of 0 and a stripe-count of 1, all files are written to OST #0, until space is exhausted. This is probably not what you meant to do. If you only want to adjust the stripe-count and keep the other parameters at their default settings, do not specify any of the other parameters:

```
lfs setstripe -c <stripe-count> <file>
```

24.3.1 Changing Striping for a Subdirectory

In a directory, the `lfs setstripe` command sets a default striping configuration for files created in the directory. The usage is the same as `lfs setstripe` for a regular file, except that the directory must exist prior to setting the default striping configuration. If a file is created in a directory with a default stripe configuration (without otherwise specifying striping), Lustre uses those striping parameters instead of the file system default for the new file.

To change the striping pattern (file layout) for a sub-directory, create a directory with desired file layout as described above. Sub-directories inherit the file layout of the root/parent directory.

Note – Striping of new files and sub-directories is done per the striping parameter settings of the root directory. Once you set striping on the root directory, then, by default, it applies to any new child directories created in that root directory (unless they have their own striping settings).

24.3.2 Using a Specific Striping Pattern/File Layout for a Single File

To use a specific striping pattern (file layout) for a specific file:

`lfs setstripe` creates a file with a given stripe pattern (file layout)

`lfs setstripe` fails if the file already exists

24.3.3 Creating a File on a Specific OST

You can use `lfs setstripe` to create a file on a specific OST. In the following example, the file "bob" will be created on the first OST (id 0).

```
$ lfs setstripe --count 1 --index 0 bob
$ dd if=/dev/zero of=bob count=1 bs=100M
1+0 records in
1+0 records out
$ lfs getstripe bob
```

OBDS:

```
0: home-OST0000_UUID ACTIVE
[...]
bob
          obdidx       objid        objid            group
          0            33459243     0x1fe8c2b        0
```

24.4 Managing Free Space

In Lustre 1.6, the MDT assigns file stripes to OSTs based on location (which OSS) and size considerations (free space) to optimize file system performance. Emptier OSTs are preferentially selected for stripes, and stripes are preferentially spread out between OSSs to increase network bandwidth utilization. The weighting factor between these two optimizations is user-adjustable.

24.4.1 Checking File System Free Space

Free space is an important consideration in assigning file stripes. The `lfs df` command shows available disk space on the mounted Lustre file system and space consumption per OST. If multiple Lustre file systems are mounted, a path may be specified, but is not required.

Option	Description
-h	Human-readable print sizes in human readable format (for example: 1K, 234M, 5G).
-i, --inodes	Lists inodes instead of block usage.

Note – The `df -i` and `lfs df -i` commands show the minimum number of inodes that can be created in the file system. Depending on the configuration, it may be possible to create more inodes than initially reported by `df -i`. Later, `df -i` operations will show the current, estimated free inode count.

If the underlying file system has fewer free blocks than inodes, then the total inode count for the file system reports only as many inodes as there are free blocks. This is done because Lustre may need to store an external attribute for each new inode, and it is better to report a free inode count that is the guaranteed, minimum number of inodes that can be created.

Examples

```
[lin-cli1] $ lfs df
UUID              1K-blockS  Used       Available  Use%   Mounted on
mds-lustre-0_UUID 9174328    1020024    8154304    11% /mnt/lustre[MDT:0]
ost-lustre-0_UUID 94181368   56330708   37850660   59% /mnt/lustre[OST:0]
ost-lustre-1_UUID 94181368   56385748   37795620   59% /mnt/lustre[OST:1]
ost-lustre-2_UUID 94181368   54352012   39829356   57% /mnt/lustre[OST:2]
filesystem summary:282544104 167068468  39829356   57% /mnt/lustre

[lin-cli1] $ lfs df -h
UUID                bytes  Used   Available  Use%       Mounted on
mds-lustre-0_UUID    8.7G   996.1M 7.8G       11%    /mnt/lustre[MDT:0]
ost-lustre-0_UUID    89.8G  53.7G  36.1G      59%    /mnt/lustre[OST:0]
ost-lustre-1_UUID    89.8G  53.8G  36.0G      59%    /mnt/lustre[OST:1]
ost-lustre-2_UUID    89.8G  51.8G  38.0G      57%    /mnt/lustre[OST:2]
filesystem summary:  269.5G 159.3G 110.1G     59%    /mnt/lustre

[lin-cli1] $ lfs df -i
UUID                Inodes     IUsed  IFree      IUse%  Mounted on
mds-lustre-0_UUID    2211572    41924  2169648    1%  /mnt/lustre[MDT:0]
ost-lustre-0_UUID    737280     12183  725097     1%  /mnt/lustre[OST:0]
ost-lustre-1_UUID    737280     12232  725048     1%  /mnt/lustre[OST:1]
ost-lustre-2_UUID    737280     12214  725066     1%  /mnt/lustre[OST:2]
filesystem summary:  2211572    41924  2169648    1%  /mnt/lustre[OST:2]
```

24.4.2 Using Stripe Allocations

There are two stripe allocation methods, round-robin and weighted. The allocation method is determined by the amount of free-space imbalance on the OSTs. The weighted allocator is used when any two OSTs are imbalanced by more than 20%. Until then, a faster round-robin allocator is used. (The round-robin order maximizes network balancing.)

24.4.3 Round-Robin Allocator

When OSTs have approximately the same amount of free space (within 20%), an efficient round-robin allocator is used. The round-robin allocator alternates stripes between OSTs on different OSSs, so the OST used for stripe 0 of each file is evenly distributed among OSTs, regardless of the stripe count. Here are several sample round-robin stripe orders (the same letter represents the different OSTs on a single OSS):

3: AAA	one 3-OST OSS
3x3: ABABAB	two 3-OST OSSs
3x4: BBABABA	one 3-OST OSS (A) and one 4-OST OSS (B)
3x5: BBABBABA	
3x5x1: BBABABABC	
3x5x2: BABABCBABC	
4x6x2: BABABCBABABC	

24.4.4 Weighted Allocator

When the free space difference between the OSTs is significant, then a weighting algorithm is used to influence OST ordering based on size and location. Note that these are weightings for a random algorithm, so the "emptiest" OST is not, necessarily, chosen every time. On average, the weighted allocator fills the emptier OSTs faster.

24.4.5 Adjusting the Weighting Between Free Space and Location

This priority can be adjusted via the `/proc/fs/lustre/lov/lustre-mdtlov/qos_prio_free proc` file. The default is 90%. Use the following command to permanently change this weighting on the MGS:

```
lctl conf_param <fsname>-MDT0000.lov.qos_prio_free=90
```

Increasing the value puts more weighting on free space. When the free space priority is set to 100%, then location is no longer used in stripe-ordering calculations, and weighting is based entirely on free space.

Note that setting the priority to 100% means that OSS distribution does not count in the weighting, but the stripe assignment is still done via a weighting—if OST2 has twice as much free space as OST1, then OST2 is twice as likely to be used, but it is not guaranteed to be used.

24.5 Handing Full OSTs

Sometimes a Lustre file system becomes unbalanced, often due to changed stripe settings. If an OST is full and an attempt is made to write more information to the file system, an error occurs. The procedures below describe how to handle a full OST.

24.5.1 Checking File System Usage

The example below shows an unbalanced file system:

```
root@LustreClient01 ~]# lfs df -h
UUID                bytes   Used  Available Use%  Mounted on
lustre-MDT0000_UUID  4.4G   214.5M   3.9G     4%   /mnt/lustre[MDT:0]
lustre-OST0000_UUID  2.0G   751.3M   1.1G    37%   /mnt/lustre[OST:0]
lustre-OST0001_UUID  2.0G   755.3M   1.1G    37%   /mnt/lustre[OST:1]
lustre-OST0002_UUID 2.0G    1.7G 155.1M   86%  /mnt/lustre[OST:2] <-
lustre-OST0003_UUID  2.0G   751.3M   1.1G    37%   /mnt/lustre[OST:3]
lustre-OST0004_UUID  2.0G   747.3M   1.1G    37%   /mnt/lustre[OST:4]
lustre-OST0005_UUID  2.0G   743.3M   1.1G    36%   /mnt/lustre[OST:5]

filesystem summary: 11.8G     5.4G    5.8G    45%  /mnt/lustre
```

In this case, OST:2 is almost full and when an attempt is made to write additional information to the file system (even with uniform striping over all the OSTs), the write command fails as follows:

```
[root@LustreClient01 ~]# lfs setstripe /mnt/lustre 4M 0 -1
[root@LustreClient01 ~]# dd if=/dev/zero of=/mnt/lustre/test_3 \
bs=10M count=100
dd: writing `/mnt/lustre/test_3': No space left on device
98+0 records in
97+0 records out
1017192448 bytes (1.0 GB) copied, 23.2411 seconds, 43.8 MB/s
```

24.5.2 Taking a Full OST Offline

To enable continued use of the file system, the full OST has to be taken offline or, more specifically, rendered read-only using the lctl command. This is done on the MDS, since the MSD allocates space for writing.

1. **Log into the MDS server:**

```
[root@LustreClient01 ~]# ssh root@192.168.0.10
root@192.168.0.10's password:
Last login: Wed Nov 26 13:35:12 2008 from 192.168.0.6
```

2. **Use the `lctl dl` command to show the status of all file system components:**

```
[root@mds ~]# lctl dl
0 UP mgs MGS MGS 9
1 UP mgc MGC192.168.0.10@tcp e384bb0e-680b-ce25-7bc9-81655dd1e813 5
2 UP mdt MDS MDS_uuid 3
3 UP lov lustre-mdtlov lustre-mdtlov_UUID 4
4 UP mds lustre-MDT0000 lustre-MDT0000_UUID 5
5 UP osc lustre-OST0000-osc lustre-mdtlov_UUID 5
6 UP osc lustre-OST0001-osc lustre-mdtlov_UUID 5
7 UP osc lustre-OST0002-osc lustre-mdtlov_UUID 5
8 UP osc lustre-OST0003-osc lustre-mdtlov_UUID 5
9 UP osc lustre-OST0004-osc lustre-mdtlov_UUID 5
10 UP osc lustre-OST0005-osc lustre-mdtlov_UUID 5
```

3. **Use `lctl deactivate` to take the full OST offline:**

```
[root@mds ~]# lctl --device 7 deactivate
```

4. **Display the status of the file system components:**

```
[root@mds ~]# lctl dl
0 UP mgs MGS MGS 9
1 UP mgc MGC192.168.0.10@tcp e384bb0e-680b-ce25-7bc9-81655dd1e813 5
2 UP mdt MDS MDS_uuid 3
3 UP lov lustre-mdtlov lustre-mdtlov_UUID 4
4 UP mds lustre-MDT0000 lustre-MDT0000_UUID 5
5 UP osc lustre-OST0000-osc lustre-mdtlov_UUID 5
6 UP osc lustre-OST0001-osc lustre-mdtlov_UUID 5
7 IN osc lustre-OST0002-osc lustre-mdtlov_UUID 5
8 UP osc lustre-OST0003-osc lustre-mdtlov_UUID 5
9 UP osc lustre-OST0004-osc lustre-mdtlov_UUID 5
10 UP osc lustre-OST0005-osc lustre-mdtlov_UUID 5
```

The device list shows that OST2 is now inactive. If a new file is now written to the file system, the write will be successful as the stripes are allocated across the remaining active OSTs.

24.5.3 Migrating Data within a File System

As stripes cannot be moved within the file system, data must be migrated manually by copying and renaming the file, removing the original file, and renaming the new file with the original file name.

1. **Identify the file(s) to be moved. In the example below, output from the getstripe command indicates that the file test_2 is located entirely on OST2:**

```
[root@LustreClient01 ~]# lfs getstripe /mnt/lustre/test_2
OBDS:
0: lustre-OST0000_UUID ACTIVE
1: lustre-OST0001_UUID ACTIVE
2: lustre-OST0002_UUID ACTIVE
3: lustre-OST0003_UUID ACTIVE
4: lustre-OST0004_UUID ACTIVE
5: lustre-OST0005_UUID ACTIVE
/mnt/lustre/test_2
obdidx      objid     objid     group
     2          8       0x8         0
```

2. **Move the file(s).**

```
[root@LustreClient01 ~]# cp /mnt/lustre/test_2 /mnt/lustre/test_2.tmp
[root@LustreClient01 ~]# rm /mnt/lustre/test_2
rm: remove regular file `/mnt/lustre/test_2'? Y
```

3. **Check the file system balance. The `df` output in the example below shows a more balanced system compared to the `df` output in the example in** Performing Direct I/O**.**

```
[root@LustreClient01 ~]# lfs df -h
UUID                    bytes   Used Available Use% Mounted on
lustre-MDT0000_UUID   4.4G  214.5M     3.9G   4% /mnt/lustre[MDT:0]
lustre-OST0000_UUID   2.0G    1.3G    598.1M  65% /mnt/lustre[OST:0]
lustre-OST0001_UUID   2.0G    1.3G    594.1M  65% /mnt/lustre[OST:1]
lustre-OST0002_UUID   2.0G  913.4M   1000.0M  45% /mnt/lustre[OST:2]
lustre-OST0003_UUID   2.0G    1.3G    602.1M  65% /mnt/lustre[OST:3]
lustre-OST0004_UUID   2.0G    1.3G    606.1M  64% /mnt/lustre[OST:4]
lustre-OST0005_UUID   2.0G    1.3G    610.1M  64% /mnt/lustre[OST:5]

filesystem summary:  11.8G    7.3G      3.9G  61% /mnt/lustre
```

4. **Change the name of the file back to the original filename so it can be found by clients.**

```
[root@LustreClient01 ~]# mv test2.tmp test2
[root@LustreClient01 ~]# ls /mnt/lustre
test1 test_2 test3 test_3 test4 test_4 test_x
```

5. **Reactivate the OST from the MDS for further writes:**

```
[root@mds ~]# lctl --device 7 activate
[root@mds ~]# lctl dl
  0 UP mgs MGS MGS 9
  1 UP mgc MGC192.168.0.10@tcp e384bb0e-680b-ce25-7bc9-816dd1e813 5
  2 UP mdt MDS MDS_uuid 3
  3 UP lov lustre-mdtlov lustre-mdtlov_UUID 4
  4 UP mds lustre-MDT0000 lustre-MDT0000_UUID 5
  5 UP osc lustre-OST0000-osc lustre-mdtlov_UUID 5
  6 UP osc lustre-OST0001-osc lustre-mdtlov_UUID 5
  7 UP osc lustre-OST0002-osc lustre-mdtlov_UUID 5
  8 UP osc lustre-OST0003-osc lustre-mdtlov_UUID 5
  9 UP osc lustre-OST0004-osc lustre-mdtlov_UUID 5
 10 UP osc lustre-OST0005-osc lustre-mdtlov_UUID
```

24.6 Creating and Managing OST Pools

Lustre 1.8 introduces the OST pool feature, which enables users to group OSTs together to make object placement more flexible. A 'pool' is the name associated with an arbitrary subset of OSTs in a Lustre cluster.

OST pools follow these rules:

- An OST can be a member of multiple pools.
- No ordering of OSTs in a pool is defined or implied.
- Stripe allocation within a pool follows the same rules as the normal stripe allocator.
- OST membership in a pool is flexible, and can change over time.

When an OST pool is defined, it can be used to allocate files. When file or directory striping is set to a pool, only OSTs in the pool are candidates for striping. If a `stripe_index` is specified which refers to an OST that is not a member of the pool, an error is returned.

OST pools are used only at file creation. If the definition of a pool changes (an OST is added or removed or the pool is destroyed), already-created files are not affected.

Note – An error (EINVAL) results if you create a file using an empty pool.

Note – Files created in a pool are not accessible from clients or servers running Lustre 1.6.5 or earlier (an error will be reported to the client). We recommend one of the following options:

- Use Lustre 1.6.6 or later prior to upgrading (to have a downgrade path available)

- Use Lustre 1.8 (without using OST pools) until there is no concern about downgrading to 1.6.5 or earlier.

24.6.1 Working with OST Pools

OST pools are defined in the configuration log on the MGS. Use the `lctl` command to:

- Create/destroy a pool
- Add/remove OSTs in a pool
- List pools and OSTs in a specific pool

The `lctl` command MUST be run on the MGS. Another requirement for managing OST pools is either to have the MDT and MGS on the same node or have a Lustre client mounted on the MGS node if it is separate from the MDS. This is needed to validate the `pool` command being run is correct.

Caution – Running the `writeconf` command on the MDS will erase all pools information (as well as any other parameters set using `lctl conf_param`). We recommend that the pools definitions (and `conf_param` settings) be executed via a script, so they can be reproduced easily after a `writeconf` is performed.

To create a new pool, run:

```
lctl pool_new <fsname>.<poolname>
```

Note – The pool name is an ASCII string up to 16 characters.

To add the named OST to a pool, run:

```
lctl pool_add <fsname>.<poolname> <ost_list>
```

Where:

`<ost_list>` is `<fsname->OST<index_range>[_UUID]`

`<index_range>` is `<ost_index_start>-<ost_index_end>[,<index_range>]` or `<ost_index_start>-<ost_index_end>/<step>`.

If the leading `<fsname>` and/or ending `_UUID` are missing, they are automatically added.

For example, to add even-numbered OSTs to `pool1` on file system `lustre`, run a single command (`add`) to add many OSTs to the pool at one time:

```
lctl pool_add lustre.pool1 OST[0-10/2]
```

Note – Each time an OST is added to a pool, a new `llog` configuration record is created. For convenience, you can run a single command.

To remove a named OST from a pool, run:

```
lctl pool_remove <fsname>.<poolname> <ost_list>
```

To destroy a pool:

```
lctl pool_destroy <fsname>.<poolname>
```

Note – All OSTs must be removed from a pool before it can be destroyed.

To list pools in the named file system, run:

```
lctl pool_list <fsname> | <pathname>
```

To list OSTs in a named pool, run:

```
lctl pool_list <fsname>.<poolname>
```

24.6.1.1 Using the lfs Command with OST Pools

Several `lfs` commands can be run with OST pools. Use the `lfs setstripe` command to associate a directory with an OST pool. This causes all new regular files and directories in the directory to be created in the pool. The `lfs` command can be used to list pools in a file system and OSTs in a named pool.

To associate a directory with a pool, so all new files and directories will be created in the pool, run:

```
lfs setstripe <filename|dirname> --pool|-p pool-name
```

To set striping patterns, run:

```
lfs setstripe [--size|-s stripe_size] [--offset|-o start_ost]
              [--count|-c stripe_count] [--pool|-p pool_name]
              <dir|filename>
```

Note – If you specifiy striping with an invalid pool name, because the pool does not exist or the pool name was mistyped, `lfs setstripe` returns an error. Re-run `lfs setstripe` and make sure the pool exists and the pool name is entered correctly.

Note – The `--pool` option for `lfs setstripe` is compatible with other modifiers. For example, you can set striping on a directory to use an explicit starting index.

24.6.2 Tips for Using OST Pools

Here are several suggestions for using OST pools.

- A directory or file can be given an extended attribute (EA), that restricts striping to a pool.
- Pools can be used to group OSTs with the same technology or performance (slower or faster), and preferred for certain jobs. Examples are SATA OSTs versus SAS OSTs or remote OSTs versus local OSTs.
- A file created in an OST pool tracks the pool by keeping the pool name in the file LOV EA.

24.7 Performing Direct I/O

Starting with 1.4.7, Lustre supports the O_DIRECT flag to open.

Applications using the read() and write() calls must supply buffers aligned on a page boundary (usually 4 K). If the alignment is not correct, the call returns -EINVAL. Direct I/O may help performance in cases where the client is doing a large amount of I/O and is CPU-bound (CPU utilization 100%).

24.7.1 Making File System Objects Immutable

An immutable file or directory is one that cannot be modified, renamed or removed. To do this:

```
chattr +i <file>
```

To remove this flag, use `chattr –i`

24.8 Other I/O Options

This section describes other I/O options, including checksums.

24.8.1 Lustre Checksums

To guard against network data corruption, a Lustre client can perform two types of data checksums: in-memory (for data in client memory) and wire (for data sent over the network). For each checksum type, a 32-bit checksum of the data read or written on both the client and server is computed, to ensure that the data has not been corrupted in transit over the network. The ldiskfs backing file system does NOT do any persistent checksumming, so it does not detect corruption of data in the OST file system.

In Lustre 1.6.5 and later, the checksumming feature is enabled, by default, on individual client nodes. If the client or OST detects a checksum mismatch, then an error is logged in the syslog of the form:

```
LustreError: BAD WRITE CHECKSUM: changed in transit before arrival at
OST: from 192.168.1.1@tcp inum 8991479/2386814769 object 1127239/0
extent [102400-106495]
```

If this happens, the client will re-read or re-write the affected data up to five times to get a good copy of the data over the network. If it is still not possible, then an I/O error is returned to the application.

To enable both types of checksums (in-memory and wire), run:

```
echo 1 > /proc/fs/lustre/llite/<fsname>/checksum_pages
```

To disable both types of checksums (in-memory and wire), run:

```
echo 0 > /proc/fs/lustre/llite/<fsname>/checksum_pages
```

To check the status of a wire checksum, run:

```
lctl get_param osc.*.checksums
```

24.8.1.1 Changing Checksum Algorithms

By default, Lustre uses the adler32 checksum algorithm, because it is robust and has a lower impact on performance than crc32. The Lustre administrator can change the checksum algorithm via /proc, depending on what is supported in the kernel.

To check which checksum algorithm is being used by Lustre, run:

```
$ cat /proc/fs/lustre/osc/<fsname>-OST<index>-osc-*/checksum_type
```

To change the wire checksum algorithm used by Lustre, run:

```
$ echo <algorithm name> /proc/fs/lustre/osc/<fsname>-OST<index>- \
osc-*/checksum_type
```

Note – The in-memory checksum always uses the adler32 algorithm, if available, and only falls back to crc32 if adler32 cannot be used.

In the following example, the `cat` command is used to determine that Lustre is using the adler32 checksum algorithm. Then the `echo` command is used to change the checksum algorithm to crc32. A second `cat` command confirms that the crc32 checksum algorithm is now in use.

```
$ cat /proc/fs/lustre/osc/lustre-OST0000-osc- \
ffff81012b2c48e0/checksum_type

crc32 [adler]

$ echo crc32 > /proc/fs/lustre/osc/lustre-OST0000-osc- \
ffff81012b2c48e0/checksum_type

$ cat /proc/fs/lustre/osc/lustre-OST0000-osc- \
ffff81012b2c48e0/checksum_type

[crc32] adler
```

24.9 Striping Using llapi

Use `llapi_file_create` to set Lustre properties for a new file. For a synopsis and description of `llapi_file_create` and examples of how to use it, see Setting Lustre Properties (man3).

You can set striping from inside programs like ioctl. To compile the sample program, you need to download `libtest.c` and `liblustreapi.c` files from the Lustre source tree.

A simple C program to demonstrate striping API – libtest.c

```
/* -*- mode: c; c-basic-offset: 8; indent-tabs-mode: nil; -*-
 * vim:expandtab:shiftwidth=8:tabstop=8:
 *
 * lustredemo - simple code examples of liblustreapi functions
 */

#include <stdio.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <dirent.h>
#include <errno.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <lustre/liblustreapi.h>
#include <lustre/lustre_user.h>
#define MAX_OSTS 1024
```

```
#define LOV_EA_SIZE(lum, num) (sizeof(*lum) + num * sizeof(*lum->lmm_objects))
#define LOV_EA_MAX(lum) LOV_EA_SIZE(lum, MAX_OSTS)

/*
This program provides crude examples of using the liblustre API functions
*/

/* Change these definitions to suit */

#define TESTDIR "/tmp"            /* Results directory */
#define TESTFILE "lustre_dummy"   /* Name for the file we create/destroy */
#define FILESIZE 262144           /* Size of the file in words */
#define DUMWORD "DEADBEEF"        /* Dummy word used to fill files */
#define MY_STRIPE_WIDTH 2         /* Set this to the number of OST required */
#define MY_LUSTRE_DIR "/mnt/lustre/ftest"


int close_file(int fd)
{
           if (close(fd) < 0) {
               fprintf(stderr, "File close failed: %d (%s)\n", errno,
strerror(errno));
               return -1;
           }
           return 0;
}

int write_file(int fd)
{
           char *stng =  DUMWORD;
           int cnt = 0;

           for( cnt = 0; cnt < FILESIZE; cnt++) {
                write(fd, stng, sizeof(stng));
           }
           return 0;
}
/* Open a file, set a specific stripe count, size and starting OST
   Adjust the parameters to suit */

int open_stripe_file()
{
           char *tfile = TESTFILE;
           int stripe_size = 65536;              /* System default is 4M */
           int stripe_offset = -1;               /* Start at default */
           int stripe_count = MY_STRIPE_WIDTH;   /*Single stripe for this
demo*/
           int stripe_pattern = 0;               /* only RAID 0 at this time
*/
           int rc, fd;
           /*
           */
           rc = llapi_file_create(tfile,
stripe_size,stripe_offset,stripe_count,stripe_pattern);
```

```
            /* result code is inverted, we may return -EINVAL or an ioctl error.
            We borrow an error message from sanity.c
            */
            if (rc) {
                 fprintf(stderr,"llapi_file_create failed: %d (%s) \n", rc,
strerror(-rc));
                 return -1;
        }
        /* llapi_file_create closes the file descriptor, we must re-open */
        fd = open(tfile, O_CREAT | O_RDWR | O_LOV_DELAY_CREATE, 0644);
        if (fd < 0) {
               fprintf(stderr, "Can't open %s file: %d (%s)\n", tfile, errno,
strerror(errno));
               return -1;
        }
        return fd;
}

/* output a list of uuids for this file */
int get_my_uuids(int fd)
{
            struct obd_uuid uuids[1024], *uuidp;      /* Output var */
            int obdcount = 1024;
            int rc,i;

            rc = llapi_lov_get_uuids(fd, uuids, &obdcount);
            if (rc != 0) {
                fprintf(stderr, "get uuids failed: %d (%s)\n",errno,
strerror(errno));
        }
        printf("This file system has %d obds\n", obdcount);
        for (i = 0, uuidp = uuids; i < obdcount; i++, uuidp++) {
               printf("UUID %d is %s\n",i, uuidp->uuid);
        }
        return 0;
}

/* Print out some LOV attributes. List our objects */
int get_file_info(char *path)
{

            struct lov_user_md *lump;
            int rc;
            int i;

            lump = malloc(LOV_EA_MAX(lump));
            if (lump == NULL) {
                return -1;
        }

        rc = llapi_file_get_stripe(path, lump);

        if (rc != 0) {
```

```
                fprintf(stderr, "get_stripe failed: %d (%s)\n",errno,
strerror(errno));
                return -1;
        }

            printf("Lov magic %u\n", lump->lmm_magic);
            printf("Lov pattern %u\n", lump->lmm_pattern);
            printf("Lov object id %llu\n", lump->lmm_object_id);
            printf("Lov object group %llu\n", lump->lmm_object_gr);
            printf("Lov stripe size %u\n", lump->lmm_stripe_size);
            printf("Lov stripe count %hu\n", lump->lmm_stripe_count);
            printf("Lov stripe offset %u\n", lump->lmm_stripe_offset);
            for (i = 0; i < lump->lmm_stripe_count; i++) {
                printf("Object index %d Objid %llu\n",
lump->lmm_objects[i].l_ost_idx, lump->lmm_objects[i].l_object_id);
        }


            free(lump);
            return rc;

}
/* Ping all OSTs that belong to this filesysem */

int ping_osts()
{
            DIR *dir;
            struct dirent *d;
            char osc_dir[100];
            int rc;

            sprintf(osc_dir, "/proc/fs/lustre/osc");
            dir = opendir(osc_dir);
            if (dir == NULL) {
                printf("Can't open dir\n");
                return -1;
            }
            while((d = readdir(dir)) != NULL) {
                if ( d->d_type == DT_DIR ) {
                    if (! strncmp(d->d_name, "OSC", 3)) {
                        printf("Pinging OSC %s ", d->d_name);
                        rc = llapi_ping("osc", d->d_name);
                        if (rc) {
                            printf("  bad\n");
                        } else {
                            printf("  good\n");
                        }
                    }
                }
            }
            return 0;

}
```

```
int main()
{
            int file;
            int rc;
            char filename[100];
            char sys_cmd[100];

            sprintf(filename, "%s/%s",MY_LUSTRE_DIR, TESTFILE);

            printf("Open a file with striping\n");
            file = open_stripe_file();
            if ( file < 0 ) {
                printf("Exiting\n");
                exit(1);




            }
            printf("Getting uuid list\n");
            rc = get_my_uuids(file);
            rintf("Write to the file\n");
            rc = write_file(file);
            rc = close_file(file);
            printf("Listing LOV data\n");
            rc = get_file_info(filename);
            printf("Ping our OSTs\n");
            rc = ping_osts();

            /* the results should match lfs getstripe */
            printf("Confirming our results with lfs getsrtipe\n");
            sprintf(sys_cmd, "/usr/bin/lfs getstripe %s/%s", MY_LUSTRE_DIR,
TESTFILE);
            system(sys_cmd);

            printf("All done\n");
            exit(rc);
}
```

Makefile for sample application:

```
gcc -g -O2 -Wall -o lustredemo libtest.c -llustreapi
clean:
rm -f core lustredemo *.o
run:
make
rm -f /mnt/lustre/ftest/lustredemo
rm -f /mnt/lustre/ftest/lustre_dummy
cp lustredemo /mnt/lustre/ftest/
```

CHAPTER 25

Lustre Security

This chapter describes Lustre security and includes the following sections:

25.1 Using ACLs

An access control list (ACL), is a set of data that informs an operating system about permissions or access rights that each user or group has to specific system objects, such as directories or files. Each object has a unique security attribute that identifies users who have access to it. The ACL lists each object and user access privileges such as read, write or execute.

25.1.1 How ACLs Work

Implementing ACLs varies between operating systems. Systems that support the Portable Operating System Interface (POSIX) family of standards share a simple yet powerful file system permission model, which should be well-known to the Linux/Unix administrator. ACLs add finer-grained permissions to this model, allowing for more complicated permission schemes. For a detailed explanation of ACLs on Linux, refer to the SuSE Labs article, **Posix Access Control Lists on Linux**:

http://www.suse.de/~agruen/acl/linux-acls/online/

We have implemented ACLs according to this model. Lustre supports the standard Linux ACL tools, setfacl, getfacl, and the historical chacl, normally installed with the ACL package.

Note – ACL support is a system-range feature, meaning that all clients have ACL enabled or not. You cannot specify which clients should enable ACL.

25.1.2 Using ACLs with Lustre

Lustre supports POSIX Access Control Lists (ACLs). An ACL consists of file entries representing permissions based on standard POSIX file system object permissions that define three classes of user (owner, group and other). Each class is associated with a set of permissions [read (r), write (w) and execute (x)].

- Owner class permissions define access privileges of the file owner.
- Group class permissions define access privileges of the owning group.
- Other class permissions define access privileges of all users not in the owner or group class.

The `ls -l` command displays the owner, group, and other class permissions in the first column of its output (for example, `-rw-r- --` for a regular file with read and write access for the owner class, read access for the group class, and no access for others).

Minimal ACLs have three entries. Extended ACLs have more than the three entries. Extended ACLs also contain a mask entry and may contain any number of named user and named group entries.

Lustre ACL support depends on the MDS, which needs to be configured to enable ACLs. Use `--mountfsoptions` to enable ACL support when creating your configuration:

```
$ mkfs.lustre --fsname spfs --mountfsoptions=acl --mdt –mgs /dev/sda
```

Alternately, you can enable ACLs at run time by using the `--acl` option with mkfs.lustre:

```
$ mount -t lustre -o acl /dev/sda /mnt/mdt
```

To check ACLs on the MDS:

```
$ lctl get_param -n mdc.home-MDT0000-mdc-*.connect_flags | grep acl
acl
```

To mount the client with no ACLs:

```
$ mount -t lustre -o noacl ibmds2@o2ib:/home /home
```

Lustre ACL support is a system-wide feature; either all clients enable ACLs or none do. Activating ACLs is controlled by MDS mount options `acl / noacl` (enable/disableACLs). Client-side mount options `acl/noacl` are ignored. You do not need to change the client configuration, and the "acl" string will not appear in the client /etc/mtab. The client acl mount option is no longer needed. If a client is mounted with that option, then this message appears in the MDS syslog:

```
...MDS requires ACL support but client does not
```

The message is harmless but indicates a configuration issue, which should be corrected.

If ACLs are not enabled on the MDS, then any attempts to reference an ACL on a client return an `Operation not supported` error.

25.1.3 Examples

These examples are taken directly from the POSIX paper referenced above. ACLs on a Lustre file system work exactly like ACLs on any Linux file system. They are manipulated with the standard tools in the standard manner. Below, we create a directory and allow a specific user access.

```
[root@client lustre]# umask 027
[root@client lustre]# mkdir rain
[root@client lustre]# ls -ld rain
drwxr-x---  2 root root 4096 Feb 20 06:50 rain
[root@client lustre]# getfacl rain
# file: rain
# owner: root
# group: root
user::rwx
group::r-x
other::---

[root@client lustre]# setfacl -m user:chirag:rwx rain
[root@client lustre]# ls -ld rain
drwxrwx---+ 2 root root 4096 Feb 20 06:50 rain
[root@client lustre]# getfacl --omit-heade rain
user::rwx
user:chirag:rwx
group::r-x
mask::rwx
other::---
```

25.2 Using Root Squash

Lustre 1.6 introduced root squash functionality, a security feature which controls super user access rights to an Lustre file system. The root squash feature works by re-mapping the user ID (UID) and group ID (GID) of the root user to a UID and GID specified by the system administrator, via the Lustre configuration management server (MGS). Additionally, the root squash feature enables the Lustre administrator to specify a NID for which UID/GID re-mapping does not apply.

Note – Before the root squash feature was added, Lustre users could run `rm -rf *` as root and remove data which should not be deleted. Using the root squash feature prevents this outcome.

25.2.1 Configuring Root Squash

Root squash functionality is managed by two configuration parameters, `rootsquash` and `nosquash_nid`.

- The `rootsquash` parameter specifies the UID and GID with which the root user accesses the Lustre file system.
- The `nosquash_nid` parameter specifies a single NID to which root squash does not apply. For example:

  ```
  nosquash_nid=172.16.245.0@tcp
  ```

  In this example, root squash does not apply to TCP clients on subnet 172.16.245.0.

25.2.2 Enabling and Tuning Root Squash

The default value for `nosquash_nid` is NULL, which means that root squashing applies to all clients. Setting the root squash UID and GID to 0 turns root squash off.

Root squash parameters can be set when the MDT is created (`mkfs.lustre --mdt`). For example:

```
mkfs.lustre --reformat --fsname=Lustre --mdt --mgs \
            --param "mdt.rootsquash=500:501" \
            --param "mdt.nosquash_nids=0@lo" /dev/sda1
```

Root squash parameters can also be changed on an unmounted device with `tunefs.lustre`. For example:

```
tunefs.lustre --param "mdt.rootsquash=65534:65534"  \
--param "mdt.nosquash_nid=192.168.0.13@tcp0" /dev/sda1
```

Root squash parameters can also be changed with the `lctl conf_param` command. For example:

```
lctl conf_param Lustre.mdt.rootsquash="1000:100"
lctl conf_param Lustre.mdt.nosquash_nid="192.168.1.1@tcp"
```

Note – When using the `lctl conf_param` command, remember that `lctl conf_param` is to be used once per a parameter.

To check root squash parameters, use the `lctl get_param` command:

```
lctl get_param mds.Lustre-MDT0000.rootsquash
lctl get_param mds.Lustre-MDT000*.nosquash_nid
```

Note – An empty `nosquash_nid` list is reported as 0@<0:0>.

25.2.3 Tips on Using Root Squash

Lustre configuration management limits root squash in several ways.

- The `lctl conf_param` value overwrites the parameter's previous value. If the new value uses an incorrect syntax, then the system continues with the old parameters and the previously-correct value is lost on remount. Be cautious when tuning root squash.
- `mkfs.lustre` and `tunefs.lustre` do not perform syntax checking. If the root squash parameters are incorrect, they are ignored on mount and the default values are used instead.
- Root squash parameters are parsed with rigorous syntax checking.
  - The `rootsquash` parameter is specified as `<decnum>':'<decnum>`.
  - The `nosquash_nids` parameter is specified as `<address>@<type><network id>`. Examples of NIDs are `192.168.1.1@tcp` and `4@elan8`.

CHAPTER 26

Lustre Operating Tips

This chapter describes tips to improve Lustre operations and includes the following sections:

26.1 Adding an OST to a Lustre File System

To add an OST to existing Lustre file system:

1. **Add a new OST by passing on the following commands, run:**

```
$ mkfs.lustre --fsname=spfs --ost --mgsnode=mds16@tcp0 /dev/sda
$ mkdir -p /mnt/test/ost0
$ mount -t lustre /dev/sda /mnt/test/ost0
```

2. **Migrate the data (possibly).**

   The file system is quite unbalanced when new empty OSTs are added. New file creations are automatically balanced. If this is a scratch file system or files are pruned at a regular interval, then no further work may be needed. Files existing prior to the expansion can be rebalanced with an in-place copy, which can be done with a simple script.

   The basic method is to copy existing files to a temporary file, then move the temp file over the old one. This should not be attempted with files which are currently being written to by users or applications. This operation redistributes the stripes over the entire set of OSTs. For a sample data migration script, see A Simple Data Migration Script.

   A very clever migration script would do the following:

   - Examine the current distribution of data.
   - Calculate how much data should move from each full OST to the empty ones.
   - Search for files on a given full OST (using `lfs getstripe`).
   - Force the new destination OST (using `lfs setstripe`).
   - Copy only enough files to address the imbalance.

   If a Lustre administrator wants to explore this approach further, per-OST disk-usage statistics can be found under `/proc/fs/lustre/osc/*/rpc_stats`

26.2 A Simple Data Migration Script

```
#!/bin/bash
# set -x

# A script to copy and check files.
# To avoid allocating objects on one or more OSTs, they should be
# deactivated on the MDS via "lctl --device {device_number}
deactivate",
# where {device_number} is from the output of "lctl dl" on the MDS.
# To guard against corruption, the file is chksum'd
# before and after the operation.
#

CKSUM=${CKSUM:-md5sum}

usage() {
    echo "usage: $0 [-O <OST_UUID-to-empty>] <dir>" 1>&2
    echo "    -O can be specified multiple times" 1>&2
    exit 1
}

while getopts "O:" opt $*; do
    case $opt in
        O) OST_PARAM="$OST_PARAM -O $OPTARG";;
        \?) usage;;
    esac
done

shift $((OPTIND - 1))
MVDIR=$1

if [ $# -ne 1 -o ! -d $MVDIR ]; then
    usage
fi

lfs find -type f $OST_PARAM $MVDIR | while read OLDNAME; do
    echo -n "$OLDNAME: "
    if [ ! -w "$OLDNAME" ]; then
        echo "No write permission, skipping"
        continue
    fi
```

```
    OLDCHK=$($CKSUM "$OLDNAME" | awk '{print $1}')
    if [ -z "$OLDCHK" ]; then
        echo "checksum error - exiting" 1>&2
          exit 1
    fi

    NEWNAME=$(mktemp "$OLDNAME.tmp.XXXXXX")
    if [ $? -ne 0 -o -z "$NEWNAME" ]; then
        echo "unable to create temp file - exiting" 1>&2
          exit 2
    fi

    cp -a "$OLDNAME" "$NEWNAME"
    if [ $? -ne 0 ]; then
        echo "copy error - exiting" 1>&2
        rm -f "$NEWNAME"
        exit 4
    fi

    NEWCHK=$($CKSUM "$NEWNAME" | awk '{print $1}')
    if [ -z "$NEWCHK" ]; then
        echo "'$NEWNAME' checksum error - exiting" 1>&2
          exit 6
    fi
    if [ $OLDCHK != $NEWCHK ]; then
       echo "'$NEWNAME' bad checksum - "$OLDNAME" not moved, exiting"
1>&2
        rm -f "$NEWNAME"
        exit 8
    else
        mv "$NEWNAME" "$OLDNAME"
        if [ $? -ne 0 ]; then
            echo "rename error - exiting" 1>&2
            rm -f "$NEWNAME"
            exit 12
        fi
    fi
    echo "done"
done
```

26.3 Adding Multiple SCSI LUNs on Single HBA

The configuration of the kernels packaged by the Lustre group is similar to that of the upstream RedHat and SuSE packages. Currently, RHEL does not enable CONFIG_SCSI_MULTI_LUN because it can cause problems with SCSI hardware.

To enable this, set the `scsi_mod max_scsi_luns=xx` option (typically, xx is 128) in either `modprobe.conf` (2.6 kernel) or `modules.conf` (2.4 kernel).

To pass this option as a kernel boot argument (in `grub.conf` or `lilo.conf`), compile the kernel with CONFIG_SCSI_MULT_LUN=y

26.4 Failures Running a Client and OST on the Same Machine

There are inherent problems if a client and OST share the same machine (and the same memory pool). An effort to relieve memory pressure (by the client), requires memory to be available to the OST. If the client is experiencing memory pressure, then the OST is as well. The OST may not get the memory it needs to help the client get the memory it needs because it is all one memory pool; this results in deadlock.

Running a client and an OST on the same machine can cause these failures:

- If the client contains a dirty file system in memory and memory pressure, a kernel thread flushes dirty pages to the file system, and it writes to a local OST. To complete the write, the OST needs to do an allocation. Then the blocking of allocation occurs while waiting for the above kernel thread to complete the write process and free up some memory. This is a deadlock condition.
- If the node with both a client and OST crashes, then the OST waits for the mounted client on that node to recover. However, since the client is now in crashed state, the OST considers it to be a new client and blocks it from mounting until the recovery completes.

As a result, running OST and client on same machine can cause a double failure and prevent a complete recovery.

26.5 Improving Lustre Metadata Performance While Using Large Directories

To improve metadata performance while using large directories, follow these tips:

- Increase RAM on the MDS – On the MDS, more memory translates into bigger caches, thereby increasing the metadata performance.
- Patch the core kernel on the MDS with the 3G/1G patch (if not running a 64-bit kernel), which increases the available kernel address space. This translates into support for bigger caches on the MDS.

PART V Reference

This part includes reference information on Lustre user utilities, configuration files and module parameters, programming interfaces, system configuration utilities, and system limits.

CHAPTER 27

User Utilities (man1)

This chapter describes user utilities and includes the following sections:

27.1 lfs

The `lfs` utility can be used for user configuration routines and monitoring. With `lfs` you can create a new file with a specific striping pattern, determine the striping pattern of existing files, and gather the extended attributes (object numbers and location) of a specific file.

Synopsis

```
lfs
lfs check <mds|osts|servers>
lfs df [-i] [-h] [path]
lfs find [[!] --atime|-A [-+]N] [[!] --mtime|-M [-+]N]
   [[!] --ctime|-C [-+]N] [--maxdepth|-D N] [--name|-n <pattern>]
   [--print|-p] [--print0|-P] [--obd|-O <uuid[s]>]
   [[!] --size|-S [+-]N[kMGTPE]] --type |-t {bcdflpsD}]
   [[!] --gid|-g|--group|-G <gname>|<gid>]
   [[!] --uid|-u|--user|-U <uname>|<uid>]
   <dirname|filename>
lfs osts
lfs getstripe [--obd|-O <uuid>] [--quiet|-q] [--verbose|-v]
              [--count|-c] [--size|-s] [--index|-i]
              [--offset|-o] [--pool|-p] [--directory|-d]
              [--recursive|-r] <dirname|filename>
lfs setstripe [--size|-s stripe-size] [--count|-c stripe-cnt]
              [--offset|-o start-ost] [--pool|-p <pool>]
              <dirname|filename>
lfs setstripe -d <dirname>
lfs poollist <filename[.<pool>] | <pathname>
lfs quota [-v][-o obd_uuid|-I ost_idx|-i mdt_idx] [-u <uname>|-u
<uid>|-g <gname>| -g <gid>] <filesystem>
lfs quota -t <-u|-g> <filesystem>
lfs quotacheck [-ugf] <filesystem>
lfs quotachown [-i] <filesystem>
lfs quotaon [-ugf] <filesystem>
lfs quotaoff [-ug] <filesystem>
lfs quotainv [-ug] [-f] <filesystem>
```

```
lfs setquota  <-u|--user|-g|--group> <uname|uid|gname|gid>
              [--block-softlimit <block-softlimit>]
              [--block-hardlimit <block-hardlimit>]
              [--inode-softlimit <inode-softlimit>]
              [--inode-hardlimit <inode-hardlimit>]
              <filesystem>
lfs setquota <-u|--user|-g|--group> <uname|uid|gname|gid>
              [-b <block-softlimit>] [-B <block-hardlimit>]
              [-i <inode-softlimit>] [-I <inode-hardlimit>]
              <filesystem>
lfs setquota -t <-u|-g>
              [--block-grace <block-grace>]
              [--inode-grace <inode-grace>]
              <filesystem>
lfs setquota -t <-u|-g>
              [-b <block-grace>] [-i <inode-grace>]
              <filesystem>
lfs help
```

Note – In the above example, the `<filesystem>` parameter refers to the mount point of the Lustre file system. The default mount point is `/mnt/lustre`.

Note – The old `lfs quota` output was very detailed and contained cluster-wide quota statistics (including cluster-wide limits for a user/group and cluster-wide usage for a user/group), as well as statistics for each MDS/OST. Now, `lfs quota` has been updated to provide only cluster-wide statistics, by default. To obtain the full report of cluster-wide limits, usage and statistics, use the `-v` option with `lfs quota`.

Description

The `lfs` utility is used to create a new file with a specific striping pattern, determine the default striping pattern, gather the extended attributes (object numbers and location) for a specific file, find files with specific attributes, list OST information, or set quota limits. It can be invoked interactively without any arguments or in a non-interactive mode with one of the supported arguments.

Options

The various `lfs` options are listed and described below. For a complete list of available options, type `help` at the `lfs` prompt.

Option	Description
check	
	Displays the status of the MDS or OSTs (as specified in the command) or all servers (MDS and OSTs).
df	
	Reports file system disk space usage or inode usage of each MDT/OST. Can limit the scope to a specific OST pool.
find	
	Searches the directory tree rooted at the given directory/filename for files that match the given parameters. The --maxdepth option limits **find** to decend at most N levels of directory tree. The --print and --print0 options print the full filename, followed by a new line or NUL character correspondingly. Using ! before an option negates its meaning (files NOT matching the parameter). Using + before a numeric value means files with the parameter OR MORE. Using - before a numeric value means files with the parameter OR LESS.
--atime	
	File was last accessed N*24 hours ago. (There is no guarantee that atime is kept coherent across the cluster.) OSTs store a transient atime that is updated when clients do read requests. Permanent **atime** is written to the MDS when the file is closed. However, on-disk atime is only updated if it is more than 60 seconds old (/proc/fs/lustre/mds/*/max_atime_diff). Lustre considers the latest **atime** from all OSTs. If a setattr is set by user, then it is updated on both the MDS and OST, allowing the **atime** to go backward.
--ctime	
	File status was last changed N*24 hours ago.
--mtime	
	File status was last modified N*24 hours ago.
--obd	
	File has an object on a specific OST(s).

Option		Description
	--size	
		File has a size in bytes or kilo-, Mega-, Giga-, Tera-, Peta- or Exabytes if a suffix is given.
	--type	
		File has a type (block, character, directory, pipe, file, symlink, socket or Door [for Solaris]).
	--uid	
		File has a specific numeric user ID.
	--user	
		File is owned by a specific user (numeric user ID is allowed).
	--gid	
		File has a specific group ID.
	--group	
		File belongs to a specific group (numeric group ID allowed).
osts		
		Lists all OSTs for the file system.
getstripe		
		Lists the striping information for a given filename or directory. By default, the stripe count, stripe size and offset are returned.
		If you only want specific striping information, then the options of **--count**,**--size**,**--index** or **--offset**, plus various combinations of these options can be used to retrieve specific information.
	--obd <uuid>	
		Lists files that have an object on a specific OST.
	--quiet	
		Lists only information about a file's object ID.
	--verbose	
		Prints additional striping information.
	--count	
		Lists the stripe count (how many OSTs to use).
	--size	
		Lists the stripe size (how much data to write to one OST before moving to the next OST).

Option		Description
	--index	
		Lists the index for each OST in the file system.
	--offset	
		Lists the OST index on which file striping starts.
	--pool	
		Lists the pools to which a file belongs.
	--directory	
		Lists entries about a specified directory instead of its contents (in the same manner as `ls -d`).
	--recursive	
		Recurses into sub-directories.
setstripe		
		Creates a new file or sets the directory default with specific striping parameters.†
	--size stripe-size*	
		Number of bytes to store on an OST before moving to the next OST. A stripe size of 0 uses the file system's default stripe size, 1MB. Can be specified with **k** (KB), **m** (MB), or **g** (GB), respectively.
	--count stripe-cnt	
		Number of OSTs over which to stripe a file. A stripe count of 0 uses the file system-wide default stripe count (1). A stripe count of -1 stripes over all available OSTs, and normally results in a file with 80 stripes.
	--offset start-ost *	
		The OST index (base 10, starting at 0) on which to start striping for this file. A **start-ost** value of -1 allows the MDS to choose the starting index. This is the default, and it means that the MDS selects the starting OST as it wants. It has no relevance on whether the MDS will use round-robin or QoS weighted allocation for the remaining stripes in the file. We strongly recommend selecting this default value, as it allows space and load balancing to be done by the MDS as needed.

Option	Description
--pool pool-name	Name of the pre-defined pool of OSTs (see lctl) that will be used for striping. The **stripe-cnt**, **stripe-size** and **start-ost** values are used as well. The **start-ost** value must be part of the pool or an error is returned.
setstripe -d <dirname>	Deletes default striping on the specified directory.
poollist <filesystem>[.<pool>] | <pathname>	Lists pools in the file system or pathname or OSTs in the file system's pool.
quota [-v] [-o obd_uuid | -i mdt_idx | -I ost_idx] [-u | -g <uname> | <uid> | <gname> | <gid>] <filesystem>	Displays disk usage and limits, either for the full file system or for objects on a specific OBD. A user or group name or an ID can be specified. If both user and group are omitted, quotas for the current UID/GID are shown. The -v option provides more verbose (with per-OBD statistics) output.
quota -t <-u | -g> <filesystem>	Displays block and inode grace times for user (-u) or group (-g) quotas.
quotacheck [-ugf] <filesystem>	Scans the specified file system for disk usage, and creates or updates quota files. Options specify quota for users (-u), groups (-g), and force (-f).
quotachown [-i] <filesystem>	Changes the file's owner and group on OSTs of the specified file system.
quotaon [-ugf] <filesystem>	Turns on file system quotas. Options specify quota for users (-u), groups (-g), and force (-f).
quotaoff [-ugf] <filesystem>	Turns off file system quotas. Options specify quota for users (-u), groups (-g), and force (-f).

Option	Description
quotainv [-ug] [-f] <filesystem>	Clears quota files (administrative quota files if used without -f, operational quota files otherwise), all of their quota entries for users (-u) or groups (-g). After running **quotainv**, you must run **quotacheck** before using quotas. **CAUTION**: Use extreme caution when using this command; its results cannot be undone.
setquota <-u|-g> <uname>|<uid>|<gname>|<gid> [--block-softlimit <block-softlimit>] [--block-hardlimit <block-hardlimit>] [--inode-softlimit <inode-softlimit>] [--inode-hardlimit <inode-hardlimit>] <filesystem>	Sets file system quotas for users or groups. Limits can be specified with --{block|inode}-{softlimit|hardlimit} or their short equivalents -b, -B, -i, -I. Users can set 1, 2, 3 or 4 limits.‡ Also, limits can be specified with special suffixes, -b, -k, -m, -g, -t, and -p to indicate units of 1, 2^10, 2^20, 2^30, 2^40 and 2^50, respectively. By default, the block limits unit is 1 kilobyte (1,024), and block limits are always kilobyte-grained (even if specified in bytes). See Examples.
setquota -t <-u|-g> [--block-grace <block-grace>] [--inode-grace <inode-grace>] <filesystem>	Sets file system quota grace times for users or groups. Grace time is specified in "XXwXXdXXhXXmXXs" format or as an integer seconds value. See Examples.
help	Provides brief help on various lfs arguments.
exit/quit	Quits the interactive lfs session.

* The default stripe-size is 0. The default stripe-start is -1. Do NOT confuse them! If you set stripe-start to 0, all new file creations occur on OST 0 (seldom a good idea).

† The file cannot exist prior to using setstripe. A directory must exist prior to using setstripe.

‡ The old setquota interface is supported, but it may be removed in a future Lustre release.

Examples

```
$ lfs setstripe -s 128k -c 2 /mnt/lustre/file1
```

Creates a file striped on two OSTs with 128 KB on each stripe.

```
$ lfs setstripe -d /mnt/lustre/dir
```

Deletes a default stripe pattern on a given directory. New files use the default striping pattern.

```
$ lfs getstripe -v /mnt/lustre/file1
```

Lists the detailed object allocation of a given file.

```
$ lfs setstripe --pool my_pool -c 2 /mnt/lustre/file
```

Creates a file striped on two OSTs from the pool `my_pool`

```
$ lfs poollist /mnt/lustre/
```

Lists the pools defined for the mounted Lustre file system `/mnt/lustre`

```
$ lfs poollist my_fs.my_pool
```

Lists the OSTs which are members of the pool `my_pool` in file system `my_fs`

```
$ lfs getstripe -v /mnt/lustre/file1
```

Lists the detailed object allocation of a given file.

```
$ lfs find /mnt/lustre
```

Efficiently lists all files in a given directory and its subdirectories.

```
$ lfs find /mnt/lustre -mtime +30 -type f -print
```

Recursively lists all regular files in a given directory more than 30 days old.

```
$ lfs find --obd OST2-UUID /mnt/lustre/
```

Recursively lists all files in a given directory that have objects on OST2-UUID. The `lfs check servers` command checks the status of all servers (MDT and OSTs).

```
$ lfs find /mnt/lustre --pool poolA
```

Finds all directories/files associated with `poolA`.

```
$ lfs find /mnt//lustre --pool ""
```

Finds all directories/files not associated with a pool.

```
$ lfs find /mnt/lustre ! --pool ""
```

Finds all directories/files associated with pool.

```
$ lfs check servers
```

Checks the status of all servers (MDT, OST)

```
$ lfs osts
```

Lists all OSTs in the file system.

```
$ lfs df -h
```

Lists space usage per OST and MDT in human-readable format.

```
$ lfs df -i
```

Lists inode usage per OST and MDT.

```
$ lfs df --pool <filesystem>[.<pool>] | <pathname>
```

List space or inode usage for a specific OST pool.

```
$ lfs quotachown -i /mnt/lustre
```

Changes file owner and group.

```
$ lfs quotacheck -ug /mnt/lustre
```

Checks quotas for user and group. Turns on quotas after making the check.

```
$ lfs quotaon -ug /mnt/lustre
```

Turns on quotas of user and group.

```
$ lfs quotaoff -ug /mnt/lustre
```

Turns off quotas of user and group.

```
$ lfs setquota -u bob --block-softlimit 2000000 --block-hardlimit
1000000 /mnt/lustre
```

Sets quotas of user 'bob', with a 1 GB block quota hardlimit and a 2 GB block quota softlimit.

```
$ lfs setquota -t -u --block-grace 1000 --inode-grace 1w4d /mnt/lustre
```

Sets grace times for user quotas: 1000 seconds for block quotas, 1 week and 4 days for inode quotas.

```
$ lfs quota -u bob /mnt/lustre
```

List quotas of user 'bob'.

```
$ lfs quota -t -u /mnt/lustre
```

Show grace times for user quotas on /mnt/lustre.

```
$ lfs setstripe --pool my_pool /mnt/lustre/dir
```

Associates a directory with the pool `my_pool`, so all new files and directories are created in the pool.

```
$ lfs find /mnt/lustre --pool poolA
```

Finds all directories/files associated with `poolA`.

```
$ lfs find /mnt//lustre --pool ""
```

Finds all directories/files not associated with a pool.

```
$ lfs find /mnt/lustre ! --pool ""
```

Finds all directories/files associated with pool.

27.2 lfsck

Lfsck ensures that objects are not referenced by multiple MDS files, that there are no orphan objects on the OSTs (objects that do not have any file on the MDS which references them), and that all of the objects referenced by the MDS exist. Under normal circumstances, Lustre maintains such coherency by distributed logging mechanisms, but under exceptional circumstances that may fail (e.g. disk failure, file system corruption leading to e2fsck repair). To avoid lengthy downtime, you can also run lfsck once Lustre is already started.

The e2fsck utility is run on each of the local MDS and OST device file systems and verifies that the underlying ldiskfs is consistent. After e2fsck is run, lfsck does distributed coherency checking for the Lustre file system. In most cases, e2fsck is sufficient to repair any file system issues and lfsck is not required.

Synopsis

```
lfsck [-c|--create] [-d|--delete] [-f|--force] [-h|--help] [-l|--
lostfound] [-n|--nofix] [-v|--verbose] --mdsdb mds_database_file --
ostdb ost1_database_file [ost2_database_file...] <filesystem>
```

Note – As shown, the <filesystem> parameter refers to the Lustre file system mount point. The default mount point is `/mnt/lustre`.

Note – For `lfsck`, database filenames must be provided as absolute pathnames. Relative paths do not work, the databases cannot be properly opened.

Options

The options and descriptions for the `lfsck` command are listed below.

Option	Description
-c	Creates (empty) missing OST objects referenced by MDS inodes.
-d	Deletes orphaned objects from the file system. Since objects on the OST are often only one of several stripes of a file, it can be difficult to compile multiple objects together in a single, usable file.
-h	Prints a brief help message.
-l	Puts orphaned objects into a lost+found directory in the root of the file system.
-n	Performs a read-only check; does not repair the file system.
-v	Verbose operation - more verbosity by specifying the option multiple times.
--mdsdb mds_database_file	MDS database file created by running e2fsck --mdsdb mds_database_file <device> on the MDS backing device. This is required.
--ostdb ost1_database_file [ost2_database_file...]	OST database files created by running e2fsck --ostdb ost_database_file <device> on each of the OST backing devices. These are required unless an OST is unavailable, in which case all objects thereon are considered missing.

Description

The lfsck utility is used to check and repair the distributed coherency of a Lustre file system. If an MDS or an OST becomes corrupt, run a distributed check on the file system to determine what sort of problems exist. Use `lfsck` to correct any defects found.

For more information on using `e2fsck` and `lfsck`, including examples, see Recovering from Errors or Corruption on a Backing File System. For information on resolving orphaned objects, see Working with Orphaned Objects.

27.3 Filefrag

The e2fsprogs package contains the filefrag tool which reports the extent of file fragmentation.

Synopsis

```
filefrag [ -belsv ] [ files...  ]
```

Description

The filefrag utility reports the extent of fragmentation in a given file. Initially, filefrag attempts to obtain extent information using FIEMAP ioctl, which is efficient and fast. If FIEMAP is not supported, then filefrag uses FIBMAP.

Note – Lustre only supports FIEMAP ioctl. FIBMAP ioctl is not supported.

In default mode[1], filefrag returns the number of physically discontiguous extents in the file. In extent or verbose mode, each extent is printed with details. For Lustre, the extents are printed in device offset order, not logical offset order.

1. The default mode is faster than the verbose/extent mode.

Options

The options and descriptions for the filefrag utility are listed below.

Option	Description
-b	Uses the 1024-byte blocksize for the output. By default, this blocksize is used by Lustre, since OSTs may use different block sizes.
-e	Uses the extent mode when printing the output.
-l	Displays extents in LUN offset order.
-s	Synchronizes the file before requesting the mapping.
--v	Uses the verbose mode when checking file fragmentation.

Examples

Lists default output.

```
$ filefrag /mnt/lustre/foo
/mnt/lustre/foo: 6 extents found
```

Lists verbose output in extent format.

```
$ filefrag  -ve /mnt/lustre/foo
Checking /mnt/lustre/foo
Filesystem type is: bd00bd0
Filesystem cylinder groups is approximately 5
File size of /mnt/lustre/foo is 157286400 (153600 blocks)
ext:device_logical:start..end physical:  start..end:length: device:flags:
0:     0..         49151:     212992..   262144:    49152:  0:      remote
1:     49152..     73727:     270336..   294912:    24576:  0:      remote
2:     73728..     76799:     24576..    27648:     3072:   0:      remote
3:     0..         57343:     196608..   253952:    57344:  1:      remote
4:     57344..     65535:     139264..   147456:    8192:   1:      remote
5:     65536..     76799:     163840..   175104:    11264:  1:      remote
/mnt/lustre/foo: 6 extents found
```

27.4 Mount

Lustre uses the standard mount(8) Linux command. When mounting a Lustre file system, mount(8) executes the `/sbin/mount.lustre` command to complete the mount. The `mount` command supports these Lustre-specific options:

Server options	Description
abort_recov	Aborts recovery when starting a target
nosvc	Starts only MGS/MGC servers
exclude	Starts with a dead OST

Client options	Description
flock	Enables/disables flock support
user_xattr/nouser_xattr	Enables/disables user-extended attributes
retry=	Number of times a client will retry to mount the file system

27.5 Handling Timeouts

Timeouts are the most common cause of hung applications. After a timeout involving an MDS or failover OST, applications attempting to access the disconnected resource wait until the connection gets established.

When a client performs any remote operation, it gives the server a reasonable amount of time to respond. If a server does not reply either due to a down network, hung server, or any other reason, a timeout occurs which requires a recovery.

If a timeout occurs, a message (similar to this one), appears on the console of the client, and in `/var/log/messages`:

```
LustreError: 26597:(client.c:810:ptlrpc_expire_one_request()) @@@ timeout

req@a2d45200 x5886/t0 o38->mds_svc_UUID@NID_mds_UUID:12 lens 168/64 ref 1 fl

RPC:/0/0 rc 0
```

CHAPTER 28

Lustre Programming Interfaces (man2)

This chapter describes public programming interfaces to control various aspects of Lustre from userspace. These interfaces are generally not guaranteed to remain unchanged over time, although we will make an effort to notify the user community well in advance of major changes. This chapter includes the following section:

- User/Group Cache Upcall

28.1 User/Group Cache Upcall

This section describes user and group upcall.

Note – For information on a universal UID/GID, see Environmental Requirements.

28.1.1 Name

Use `/proc/fs/lustre/mds/mds-service/group_upcall` to look up a given user's group membership.

28.1.2 Description

The group upcall file contains the path to an executable that, when properly installed, is invoked to resolve a numeric UID to a group membership list. This utility should complete the `mds_grp_downcall_data` data structure (see Data structures) and write it to the `/proc/fs/lustre/mds/mds-service/group_info` pseudo-file.

For a sample upcall program, see `lustre/utils/l_getgroups.c` in the Lustre source distribution.

28.1.2.1 Primary and Secondary Groups

The mechanism for the primary/secondary group is as follows:

- The MDS issues an upcall (set per MDS) to map the numeric UID to the supplementary group(s).
- If there is no upcall or if there is an upcall and it fails, supplementary groups will be added as supplied by the client (as they are now).
- The default upcall is `/usr/sbin/l_getgroups`, which uses the Lustre group-supplied upcall. It looks up the UID in `/etc/passwd`, and if it finds the UID, it looks for supplementary groups in `/etc/group` for that username. You are free to enhance `l_getgroups` to look at an external database for supplementary groups information.
- The default group upcall is set by mkfs.lustre. To set the upcall, use `echo {path} > /proc/fs/lustre/mds/{mdsname}/group_upcall` or `tunefs.lustre --param`.
- To avoid repeated upcalls, the supplementary group information is cached by the MDS. The default cache time is 300 seconds, but can be changed via `/proc/fs/lustre/mds/{mdsname}/group_expire`. The kernel waits, at most, 5 seconds (by default, `/proc/fs/lustre/mds/{mdsname}/group_acquire_expire changes`) for the upcall to complete and will take the "failure" behavior as described above. It is possible to flush cached entries by writing to the `/proc/fs/lustre/mds/{mdsname}/group_flush` file.

28.1.3 Parameters

- Name of the MDS service
- Numeric UID

28.1.4 Data structures

```
#include <lustre/lustre_user.h>
#define MDS_GRP_DOWNCALL_MAGIC 0x6d6dd620
struct mds_grp_downcall_data {
     __u32    mgd_magic;
     __u32    mgd_err;
     __u32    mgd_uid;
     __u32    mgd_gid;
     __u32    mgd_ngroups;
     __u32    mgd_groups[0];
};
```

CHAPTER 29

Setting Lustre Properties (man3)

This chapter describes how to use `llapi` to set Lustre file properties.

29.1 Using llapi

Several llapi commands are available to set Lustre properties, `llapi_file_create`, `llapi_file_get_stripe`, and `llapi_file_open`. These commands are described in the following sections:

29.1.1 llapi_file_create

Use `llapi_file_create` to set Lustre properties for a new file.

Synopsis

```
#include <lustre/liblustreapi.h>
#include <lustre/lustre_user.h>

int llapi_file_create(char *name, long stripe_size,
int stripe_offset, int stripe_count, int stripe_pattern);
```

Description

The `llapi_file_create()` function sets a file descriptor's Lustre striping information. The file descriptor is then accessed with open ().

Option	Description
llapi_file_create()	If the file already exists, this parameter returns to 'EEXIST'. If the stripe parameters are invalid, this parameter returns to 'EINVAL'.
stripe_size	This value must be an even multiple of system page size, as shown by getpagesize (). The default Lustre stripe size is 4MB.
stripe_offset	Indicates the starting OST for this file.
stripe_count	Indicates the number of OSTs that this file will be striped across.
stripe_pattern	Indicates the RAID pattern.

Note – Currently, only RAID 0 is supported. To use the system defaults, set these values: stripe_size = 0, stripe_offset = -1, stripe_count = 0, stripe_pattern = 0

Examples

System default size is 4 MB.

```
char *tfile = TESTFILE;
int stripe_size = 65536
```

To start at default, run:

```
int stripe_offset = -1
```

To start at the default, run:

```
int stripe_count = 1
```

To set a single stripe for this example, run:

```
int stripe_pattern = 0
```

Currently, only RAID 0 is supported.

```
int stripe_pattern = 0;
int rc, fd;
rc = llapi_file_create(tfile,
stripe_size,stripe_offset, stripe_count,stripe_pattern);
```

Result code is inverted, you may return with 'EINVAL' or an ioctl error.

```
if (rc) {
fprintf(stderr,"llapi_file_create failed: %d (%s) 0, rc,
strerror(-rc));
return -1;
}
```

`llapi_file_create` closes the file descriptor. You must re-open the descriptor. To do this, run:

```
fd = open(tfile, O_CREAT | O_RDWR | O_LOV_DELAY_CREATE, 0644);
if (fd < 0) \ {
fprintf(stderr, "Can't open %s file: %s0, tfile,
str-
error(errno));
return -1;
}
```

29.1.2 llapi_file_get_stripe

Use `llapi_file_get_stripe` to get striping information.

Synopsis

```
int llapi_file_get_stripe(const char *path, struct lov_user_md *lum)
```

Description

The `llapi_file_get_stripe` function returns the striping information to the caller. If it returns a zero (0), the operation was successful; a negative number means there was a failure.

Option	Description
path	The path of the file.
lum	The returned striping information.
return	A value of zero (0) mean the operation was successful. A value of a negative number means there was a failure.
stripe_count	Indicates the number of OSTs that this file will be striped across.
stripe_pattern	Indicates the RAID pattern.

29.1.3 llapi_file_open

The `llapi_file_open` command opens or creates a file with the specified striping parameters.

Synopsis

```
int llapi_file_open(const char *name, int flags, int mode, unsigned
long stripe_size, int stripe_offset, int stripe_count, int
stripe_pattern)
```

Description

The `llapi_file_open` function opens or creates a file with the specified striping parameters. If it returns a zero (0), the operation was successful; a negative number means there was a failure.

Option	Description
name	The name of the file.
flags	This opens flags.
mode	This opens modes.
stripe_size	The stripe size of the file.
stripe_offset	The stripe offset (stripe_index) of the file.
stripe_count	The stripe count of the file.
stripe_pattern	The stripe pattern of the file.

29.1.4 llapi_quotactl

Use `llapi_quotactl` to manipulate disk quotas on a Lustre file system.

Synopsis

```
#include <liblustre.h>
#include <lustre/lustre_idl.h>
#include <lustre/liblustreapi.h>
#include <lustre/lustre_user.h>
int llapi_quotactl(char" " *mnt," " struct if_quotactl" " *qctl)

struct if_quotactl {
        __u32                   qc_cmd;
        __u32                   qc_type;
        __u32                   qc_id;
        __u32                   qc_stat;
        struct obd_dqinfo       qc_dqinfo;
        struct obd_dqblk        qc_dqblk;
        char                    obd_type[16];
        struct obd_uuid         obd_uuid;
};
struct obd_dqblk {
        __u64 dqb_bhardlimit;
        __u64 dqb_bsoftlimit;
        __u64 dqb_curspace;
        __u64 dqb_ihardlimit;
        __u64 dqb_isoftlimit;
        __u64 dqb_curinodes;
        __u64 dqb_btime;
        __u64 dqb_itime;
        __u32 dqb_valid;
        __u32 padding;
};
struct obd_dqinfo {
        __u64 dqi_bgrace;
        __u64 dqi_igrace;
        __u32 dqi_flags;
        __u32 dqi_valid;
};
struct obd_uuid {
        char uuid[40];
};
```

Description

The `llapi_quotactl()` command manipulates disk quotas on a Lustre file system mount. qc_cmd indicates a command to be applied to UID qc_id or GID qc_id.

Option	Description
LUSTRE_Q_QUOTAON	Turns on quotas for a Lustre file system. qc_type is USRQUOTA, GRPQUOTA or UGQUOTA (both user and group quota). The quota files must exist. They are normally created with the llapi_quotacheck(3) call. This call is restricted to the super user privilege.
LUSTRE_Q_QUOTAOFF	Turns off quotas for a Lustre file system. qc_type is USRQUOTA, GRPQUOTA or UGQUOTA (both user and group quota). This call is restricted to the super user privilege.
LUSTRE_Q_GETQUOTA	Gets disk quota limits and current usage for user or group qc_id. qc_type is USRQUOTA or GRPQUOTA. UUID may be filled with OBD UUID string to query quota information from a specific node. dqb_valid may be set nonzero to query information only from MDS. If UUID is an empty string and dqb_valid is zero then cluster-wide limits and usage are returned. On return, obd_dqblk contains the requested information (block limits unit is kilobyte). Quotas must be turned on before using this command.
LUSTRE_Q_SETQUOTA	Sets disk quota limits for user or group qc_id. qc_type is USRQUOTA or GRPQUOTA. dqb_valid must be set to QIF_ILIMITS, QIF_BLIMITS or QIF_LIMITS (both inode limits and block limits) dependent on updating limits. obd_dqblk must be filled with limits values (as set in dqb_valid, block limits unit is kilobyte). Quotas must be turned on before using this command.
LUSTRE_Q_GETINFO	Gets information about quotas. qc_type is either USRQUOTA or GRPQUOTA. On return, dqi_igrace is inode grace time (in seconds), dqi_bgrace is block grace time (in seconds), dqi_flags is not used by the current Lustre version.
LUSTRE_Q_SETINFO	Sets quota information (like grace times). qc_type is either USRQUOTA or GRPQUOTA. dqi_igrace is inode grace time (in seconds), dqi_bgrace is block grace time (in seconds), dqi_flags is not used by the current Lustre version and must be zeroed.

Return Values

`llapi_quotactl()` returns:

0 on success

-1 on failure and sets error number to indicate the error

llapi Errors

llapi errors are described below.

Errors	Description
EFAULT	qctl is invalid.
ENOSYS	Kernel or Lustre modules have not been compiled with the QUOTA option.
ENOMEM	Insufficient memory to complete operation.
ENOTTY	qc_cmd is invalid.
EBUSY	Cannot process during quotacheck.
ENOENT	UUID does not correspond to OBD or mnt does not exist.
EPERM	The call is privileged and the caller is not the super user.
ESRCH	No disk quota is found for the indicated user. Quotas have not been turned on for this file system.

29.1.5 llapi_path2fid

Use `llapi_path2fid` to get the FID from the pathname.

Synopsis

```
#include <lustre/liblustreapi.h>
#include <lustre/lustre_user.h>

int llapi_path2fid(const char *path, unsigned long long *seq,
unsigned long *oid, unsigned long *ver)
```

Description

The `llapi_path2fid` function returns the FID (sequence : object ID : version) for the pathname.

Return Values

`llapi_path2fid` returns:

0	on success
non-zero value	on failure

CHAPTER 30

Configuration Files and Module Parameters (man5)

This section describes configuration files and module parameters and includes the following sections:

30.1 Introduction

LNET network hardware and routing are now configured via module parameters. Parameters should be specified in the `/etc/modprobe.conf` file, for example:

```
alias lustre llite
options lnet networks=tcp0,elan0
```

The above option specifies that this node should use all the available TCP and Elan interfaces.

Module parameters are read when the module is first loaded. Type-specific LND modules (for instance, ksocklnd) are loaded automatically by the LNET module when LNET starts (typically upon modprobe ptlrpc).

Under Linux 2.6, LNET configuration parameters can be viewed under `/sys/module/`; generic and acceptor parameters under LNET, and LND-specific parameters under the name of the corresponding LND.

Under Linux 2.4, sysfs is not available, but the LND-specific parameters are accessible via equivalent paths under `/proc`.

Important: All old (pre v.1.4.6) Lustre configuration lines should be removed from the module configuration files and replaced with the following. Make sure that CONFIG_KMOD is set in your linux.config so LNET can load the following modules it needs. The basic module files are:

`modprobe.conf` (for Linux 2.6)

```
alias lustre llite
options lnet networks=tcp0,elan0
```

`modules.conf` (for Linux 2.4)

```
alias lustre llite
options lnet networks=tcp0,elan0
```

For the following parameters, default option settings are shown in parenthesis. Changes to parameters marked with a W affect running systems. (Unmarked parameters can only be set when LNET loads for the first time.) Changes to parameters marked with Wc only have effect when connections are established (existing connections are not affected by these changes.)

30.2 Module Options

- With routed or other multi-network configurations, use ip2nets rather than networks, so all nodes can use the same configuration.
- For a routed network, use the same "routes" configuration everywhere. Nodes specified as routers automatically enable forwarding and any routes that are not relevant to a particular node are ignored. Keep a common configuration to guarantee that all nodes have consistent routing tables.
- A separate modprobe.conf.lnet included from modprobe.conf makes distributing the configuration much easier.
- If you set `config_on_load=1`, LNET starts at modprobe time rather than waiting for Lustre to start. This ensures routers start working at module load time.

  ```
  # lctl
  # lctl> net down
  ```

- Remember the `lctl ping {nid}` command - it is a handy way to check your LNET configuration.

30.2.1 LNET Options

This section describes LNET options.

30.2.1.1 Network Topology

Network topology module parameters determine which networks a node should join, whether it should route between these networks, and how it communicates with non-local networks.

Here is a list of various networks and the supported software stacks:

Network	Software Stack
openib	OpenIB gen1/Mellanox Gold
iib	Silverstorm (Infinicon)
vib	Voltaire
o2ib	OpenIB gen2
cib	Cisco
mx	Myrinet MX
gm	Myrinet GM-2
elan	Quadrics QSNet

Note – Lustre ignores the loopback interface (lo0), but Lustre use any IP addresses aliased to the loopback (by default). When in doubt, explicitly specify networks.

ip2nets ("") is a string that lists globally-available networks, each with a set of IP address ranges. LNET determines the locally-available networks from this list by matching the IP address ranges with the local IPs of a node. The purpose of this option is to be able to use the same modules.conf file across a variety of nodes on different networks. The string has the following syntax.

```
<ip2nets> :== <net-match> [ <comment> ] { <net-sep> <net-match> }
<net-match> :== [ <w> ] <net-spec> <w> <ip-range> { <w> <ip-range> }
[ <w> ]
<net-spec> :== <network> [ "(" <interface-list> ")" ]
<network> :== <nettype> [ <number> ]
<nettype> :== "tcp" | "elan" | "openib" | ...
<iface-list> :== <interface> [ "," <iface-list> ]
<ip-range> :== <r-expr> "." <r-expr> "." <r-expr> "." <r-expr>
<r-expr> :== <number> | "*" | "[" <r-list> "]"
<r-list> :== <range> [ "," <r-list> ]
<range> :== <number> [ "-" <number> [ "/" <number> ] ]
<comment :== "#" { <non-net-sep-chars> }
<net-sep> :== ";" | "\n"
<w> :== <whitespace-chars> { <whitespace-chars> }
```

<**net-spec**> contains enough information to uniquely identify the network and load an appropriate LND. The LND determines the missing "address-within-network" part of the NID based on the interfaces it can use.

<**iface-list**> specifies which hardware interface the network can use. If omitted, all interfaces are used. LNDs that do not support the <iface-list> syntax cannot be configured to use particular interfaces and just use what is there. Only a single instance of these LNDs can exist on a node at any time, and <iface-list> must be omitted.

<**net-match**> entries are scanned in the order declared to see if one of the node's IP addresses matches one of the <ip-range> expressions. If there is a match, <net-spec> specifies the network to instantiate. Note that it is the first match for a particular network that counts. This can be used to simplify the match expression for the general case by placing it after the special cases. For example:

```
ip2nets="tcp(eth1,eth2) 134.32.1.[4-10/2]; tcp(eth1) *.*.*.*"
```

4 nodes on the 134.32.1.* network have 2 interfaces (134.32.1.{4,6,8,10}) but all the rest have 1.

```
ip2nets="vib 192.168.0.*; tcp(eth2) 192.168.0.[1,7,4,12]"
```

This describes an IB cluster on 192.168.0.*. Four of these nodes also have IP interfaces; these four could be used as routers.

Note that match-all expressions (For instance, *.*.*.*) effectively mask all other <*net-match*> entries specified after them. They should be used with caution.

Here is a more complicated situation, the route parameter is explained below. We have:

- Two TCP subnets
- One Elan subnet
- One machine set up as a router, with both TCP and Elan interfaces
- IP over Elan configured, but only IP will be used to label the nodes.

```
options lnet ip2nets="tcp198.129.135.* 192.128.88.98; \
             elan  198.128.88.98 198.129.135.3;" \
             routes="tcp  1022@elan# Elan NID of router;\
             elan  198.128.88.98@tcp # TCP NID of router  "
```

30.2.1.2 networks ("tcp")

This is an alternative to "ip2nets" which can be used to specify the networks to be instantiated explicitly. The syntax is a simple comma separated list of <net-spec>s (see above). The default is only used if neither “ip2nets” nor “networks” is specified.

30.2.1.3 routes (“”)

This is a string that lists networks and the NIDs of routers that forward to them.

It has the following syntax (<w> is one or more whitespace characters):

```
<routes> :== <route>{ ; <route> }
<route> :== [<net>[<w><hopcount>]<w><nid>{<w><nid>}
```

So a node on the network tcp1 that needs to go through a router to get to the Elan network:

```
options lnet networks=tcp1 routes="elan 1 192.168.2.2@tcp1"
```

The hopcount is used to help choose the best path between multiply-routed configurations.

A simple but powerful expansion syntax is provided, both for target networks and router NIDs as follows.

```
<expansion> :== "[" <entry> { "," <entry> } "]"
<entry> :== <numeric range> | <non-numeric item>
<numeric range> :== <number> [ "-" <number> [ "/" <number> ] ]
```

The expansion is a list enclosed in square brackets. Numeric items in the list may be a single number, a contiguous range of numbers, or a strided range of numbers. For example, routes="elan 192.168.1.[22-24]@tcp" says that network elan0 is adjacent (hopcount defaults to 1); and is accessible via 3 routers on the tcp0 network (192.168.1.22@tcp, 192.168.1.23@tcp and 192.168.1.24@tcp).

routes="[tcp,vib] 2 [8-14/2]@elan" says that 2 networks (tcp0 and vib0) are accessible through 4 routers (8@elan, 10@elan, 12@elan and 14@elan). The hopcount of 2 means that traffic to both these networks will be traversed 2 routers - first one of the routers specified in this entry, then one more.

Duplicate entries, entries that route to a local network, and entries that specify routers on a non-local network are ignored.

Equivalent entries are resolved in favor of the route with the shorter hopcount. The hopcount, if omitted, defaults to 1 (the remote network is adjacent).

It is an error to specify routes to the same destination with routers on different local networks.

If the target network string contains no expansions, then the hopcount defaults to 1 and may be omitted (that is, the remote network is adjacent). In practice, this is true for most multi-network configurations. It is an error to specify an inconsistent hop count for a given target network. This is why an explicit hopcount is required if the target network string specifies more than one network.

30.2.1.4 forwarding ("")

This is a string that can be set either to "enabled" or "disabled" for explicit control of whether this node should act as a router, forwarding communications between all local networks.

A standalone router can be started by simply starting LNET ("modprobe ptlrpc") with appropriate network topology options.

Variable	Description
acceptor	The acceptor is a TCP/IP service that some LNDs use to establish communications. If a local network requires it and it has not been disabled, the acceptor listens on a single port for connection requests that it redirects to the appropriate local network. The acceptor is part of the LNET module and configured by the following options: • **secure** - Accept connections only from reserved TCP ports (< 1023). • **all** - Accept connections from any TCP port. NOTE: this is required for liblustre clients to allow connections on non-privileged ports. • **none** - Do not run the acceptor.
accept_port (988)	Port number on which the acceptor should listen for connection requests. All nodes in a site configuration that require an acceptor must use the same port.
accept_backlog (127)	Maximum length that the queue of pending connections may grow to (see listen(2)).
accept_timeout (5, W)	Maximum time in seconds the acceptor is allowed to block while communicating with a peer.
accept_proto_version	Version of the acceptor protocol that should be used by outgoing connection requests. It defaults to the most recent acceptor protocol version, but it may be set to the previous version to allow the node to initiate connections with nodes that only understand that version of the acceptor protocol. The acceptor can, with some restrictions, handle either version (that is, it can accept connections from both 'old' and 'new' peers). For the current version of the acceptor protocol (version 1), the acceptor is compatible with old peers if it is only required by a single local network.

30.2.2 SOCKLND Kernel TCP/IP LND

The SOCKLND kernel TCP/IP LND (socklnd) is connection-based and uses the acceptor to establish communications via sockets with its peers.

It supports multiple instances and load balances dynamically over multiple interfaces. If no interfaces are specified by the ip2nets or networks module parameter, all non-loopback IP interfaces are used. The address-within-network is determined by the address of the first IP interface an instance of the socklnd encounters.

Consider a node on the “edge” of an InfiniBand network, with a low-bandwidth management Ethernet (eth0), IP over IB configured (ipoib0), and a pair of GigE NICs (eth1,eth2) providing off-cluster connectivity. This node should be configured with "networks=vib,tcp(eth1,eth2)” to ensure that the socklnd ignores the management Ethernet and IPoIB.

Variable	Description
timeout (50,W)	Time (in seconds) that communications may be stalled before the LND completes them with failure.
nconnds (4)	Sets the number of connection daemons.
min_reconnectms (1000,W)	Minimum connection retry interval (in milliseconds). After a failed connection attempt, this is the time that must elapse before the first retry. As connections attempts fail, this time is doubled on each successive retry up to a maximum of 'max_reconnectms'.
max_reconnectms (6000,W)	Maximum connection retry interval (in milliseconds).
eager_ack (0 on linux, 1 on darwin,W)	Boolean that determines whether the socklnd should attempt to flush sends on message boundaries.
typed_conns (1,Wc)	Boolean that determines whether the socklnd should use different sockets for different types of messages. When clear, all communication with a particular peer takes place on the same socket. Otherwise, separate sockets are used for bulk sends, bulk receives and everything else.
min_bulk (1024,W)	Determines when a message is considered "bulk".
tx_buffer_size, rx_buffer_size (8388608,Wc)	Socket buffer sizes. Setting this option to zero (0), allows the system to auto-tune buffer sizes. WARNING: Be very careful changing this value as improper sizing can harm performance.
nagle (0,Wc)	Boolean that determines if nagle should be enabled. It should never be set in production systems.

Variable	Description
keepalive_idle (30,Wc)	Time (in seconds) that a socket can remain idle before a keepalive probe is sent. Setting this value to zero (0) disables keepalives.
keepalive_intvl (2,Wc)	Time (in seconds) to repeat unanswered keepalive probes. Setting this value to zero (0) disables keepalives.
keepalive_count (10,Wc)	Number of unanswered keepalive probes before pronouncing socket (hence peer) death.
enable_irq_affinity (0,Wc)	Boolean that determines whether to enable IRQ affinity. The default is zero (0). When set, socklnd attempts to maximize performance by handling device interrupts and data movement for particular (hardware) interfaces on particular CPUs. This option is not available on all platforms. This option requires an SMP system to exist and produces best performance with multiple NICs. Systems with multiple CPUs and a single NIC may see increase in the performance with this parameter disabled.
zc_min_frag (2048,W)	Determines the minimum message fragment that should be considered for zero-copy sends. Increasing it above the platform's PAGE_SIZE disables all zero copy sends. This option is not available on all platforms.

30.2.3 QSW LND

The QSW LND (qswlnd) is connection-less and, therefore, does not need the acceptor. It is limited to a single instance, which uses all Elan "rails" that are present and dynamically load balances over them.

The address-with-network is the node's Elan ID. A specific interface cannot be selected in the "networks" module parameter.

Variable	Description
tx_maxcontig (1024)	Integer that specifies the maximum message payload (in bytes) to copy into a pre-mapped transmit buffer
mtxmsgs (8)	Number of "normal" message descriptors for locally-initiated communications that may block for memory (callers block when this pool is exhausted).
nnblk_txmsg (512 with a 4K page size, 256 otherwise)	Number of "reserved" message descriptors for communications that may not block for memory. This pool must be sized large enough so it is never exhausted.
nrxmsg_small (256)	Number of "small" receive buffers to post (typically everything apart from bulk data).
ep_envelopes_small (2048)	Number of message envelopes to reserve for the "small" receive buffer queue. This determines a breakpoint in the number of concurrent senders. Below this number, communication attempts are queued, but above this number, the pre-allocated envelope queue will fill, causing senders to back off and retry. This can have the unfortunate side effect of starving arbitrary senders, who continually find the envelope queue is full when they retry. This parameter should therefore be increased if envelope queue overflow is suspected.
nrxmsg_large (64)	Number of "large" receive buffers to post (typically for routed bulk data).
ep_envelopes_large (256)	Number of message envelopes to reserve for the "large" receive buffer queue. For more information on message envelopes, see the ep_envelopes_small option (above).
optimized_puts (32768,W)	Smallest non-routed PUT that will be RDMA'd.
optimized_gets (1,W)	Smallest non-routed GET that will be RDMA'd.

30.2.4 RapidArray LND

The RapidArray LND (ralnd) is connection-based and uses the acceptor to establish connections with its peers. It is limited to a single instance, which uses all (both) RapidArray devices present. It load balances over them using the XOR of the source and destination NIDs to determine which device to use for communication.

The address-within-network is determined by the address of the single IP interface that may be specified by the "networks" module parameter. If this is omitted, then the first non-loopback IP interface that is up is used instead.

Variable	Description
n_connd (4)	Sets the number of connection daemons.
min_reconnect_interval (1,W)	Minimum connection retry interval (in seconds). After a failed connection attempt, this sets the time that must elapse before the first retry. As connections attempts fail, this time is doubled on each successive retry, up to a maximum of the max_reconnect_interval option.
max_reconnect_interval (60,W)	Maximum connection retry interval (in seconds).
timeout (30,W)	Time (in seconds) that communications may be stalled before the LND completes them with failure.
ntx (64)	Number of "normal" message descriptors for locally-initiated communications that may block for memory (callers block when this pool is exhausted).
ntx_nblk (256)	Number of "reserved" message descriptors for communications that may not block for memory. This pool must be sized large enough so it is never exhausted.
fma_cq_size (8192)	Number of entries in the RapidArray FMA completion queue to allocate. It should be increased if the ralnd starts to issue warnings that the FMA CQ has overflowed. This is only a performance issue.
max_immediate (2048,W)	Size (in bytes) of the smallest message that will be RDMA'd, rather than being included as immediate data in an FMA. All messages greater than 6912 bytes must be RDMA'd (FMA limit).

30.2.5 VIB LND

The VIB LND is connection-based, establishing reliable queue-pairs over InfiniBand with its peers. It does not use the acceptor. It is limited to a single instance, using a single HCA that can be specified via the "networks" module parameter. If this is omitted, it uses the first HCA in numerical order it can open. The address-within-network is determined by the IPoIB interface corresponding to the HCA used.

Variable	Description
service_number (0x11b9a2)	Fixed IB service number on which the LND listens for incoming connection requests. NOTE: All instances of the viblnd on the same network must have the same setting for this parameter.
arp_retries (3,W)	Number of times the LND will retry ARP while it establishes communications with a peer.
min_reconnect_interval (1,W)	Minimum connection retry interval (in seconds). After a failed connection attempt, this sets the time that must elapse before the first retry. As connections attempts fail, this time is doubled on each successive retry, up to a maximum of the max_reconnect_interval option.
max_reconnect_interval (60,W)	Maximum connection retry interval (in seconds).
timeout (50,W)	Time (in seconds) that communications may be stalled before the LND completes them with failure.
ntx (32)	Number of "normal" message descriptors for locally-initiated communications that may block for memory (callers block when this pool is exhausted).
ntx_nblk (256	Number of "reserved" message descriptors for communications that may not block for memory. This pool must be sized large enough so it is never exhausted.
concurrent_peers (1152)	Maximum number of queue pairs and, therefore, the maximum number of peers that the instance of the LND may communicate with.
hca_basename ("InfiniHost")	Used to construct HCA device names by appending the device number.
ipif_basename ("ipoib")	Used to construct IPoIB interface names by appending the same device number as is used to generate the HCA device name.
local_ack_timeout (0x12,Wc)	Used to construct IPoIB interface names by appending the same device number as is used to generate the HCA device name.
retry_cnt (7,Wc)	Low-level QP parameter. Only change it from the default value if so advised.

Variable	Description
rnr_cnt (6,Wc)	Low-level QP parameter. Only change it from the default value if so advised.
rnr_nak_timer (0x10,Wc)	Low-level QP parameter. Only change it from the default value if so advised.
fmr_remaps (1000)	Controls how often FMR mappings may be reused before they must be unmapped. Only change it from the default value if so advised
cksum (0,W)	Boolean that determines if messages (NB not RDMAs) should be check-summed. This is a diagnostic feature that should not normally be enabled.

30.2.6 OpenIB LND

The OpenIB LND is connection-based and uses the acceptor to establish reliable queue-pairs over InfiniBand with its peers. It is limited to a single instance that uses only IB device '0'.

The address-within-network is determined by the address of the single IP interface that may be specified by the "networks" module parameter. If this is omitted, the first non-loopback IP interface that is up, is used instead. It uses the acceptor to establish connections with its peers.

Variable	Description
n_connd (4)	Sets the number of connection daemons. The default value is 4.
min_reconnect_interval (1,W)	Minimum connection retry interval (in seconds). After a failed connection attempt, this sets the time that must elapse before the first retry. As connections attempts fail, this time is doubled on each successive retry, up to a maximum of 'max_reconnect_interval'.
max_reconnect_interval (60,W)	Maximum connection retry interval (in seconds).
timeout (50,W)	Time (in seconds) that communications may be stalled before the LND completes them with failure.
ntx (64)	Number of "normal" message descriptors for locally-initiated communications that may block for memory (callers block when this pool is exhausted).
ntx_nblk (256)	Number of "reserved" message descriptors for communications that may not block for memory. This pool must be sized large enough so it is never exhausted.
concurrent_peers (1024)	Maximum number of queue pairs and, therefore, the maximum number of peers that the instance of the LND may communicate with.
cksum (0,W)	Boolean that determines whether messages (NB not RDMAs) should be check-summed. This is a diagnostic feature that should not normally be enabled.

30.2.7 Portals LND (Linux)

The Portals LND Linux (ptllnd) can be used as a interface layer to communicate with Sandia Portals networking devices. This version is intended to work on Cray XT3 Linux nodes that use Cray Portals as a network transport.

Message Buffers

When ptllnd starts up, it allocates and posts sufficient message buffers to allow all expected peers (set by `concurrent_peers`) to send one unsolicited message. The first message that a peer actually sends is a

(so-called) "HELLO" message, used to negotiate how much additional buffering to setup (typically 8 messages). If 10000 peers actually exist, then enough buffers are posted for 80000 messages.

The maximum message size is set by the `max_msg_size` module parameter (default value is 512). This parameter sets the bulk transfer breakpoint. Below this breakpoint, payload data is sent in the message itself. Above this breakpoint, a buffer descriptor is sent and the receiver gets the actual payload.

The buffer size is set by the rxb_npages module parameter (default value is 1). The default conservatively avoids allocation problems due to kernel memory fragmentation. However, increasing this value to 2 is probably not risky.

The ptllnd also keeps an additional rxb_nspare buffers (default value is 8) posted to account for full buffers being handled.

Assuming a 4K page size with 10000 peers, 1258 buffers can be expected to be posted at startup, increasing to a maximum of 10008 as peers that are actually connected. By doubling `rxb_npages` halving `max_msg_size`, this number can be reduced by a factor of 4.

ME/MD Queue Length

The ptllnd uses a single portal set by the portal module parameter (default value of 9) for both message and bulk buffers. Message buffers are always attached with PTL_INS_AFTER and match anything sent with "message" matchbits. Bulk buffers are always attached with PTL_INS_BEFORE and match only specific matchbits for that particular bulk transfer.

This scheme assumes that the majority of ME / MDs posted are for "message" buffers, and that the overhead of searching through the preceding "bulk" buffers is acceptable. Since the number of "bulk" buffers posted at any time is also dependent on the bulk transfer breakpoint set by `max_msg_size`, this seems like an issue worth measuring at scale.

TX Descriptors

The ptllnd has a pool of so-called "tx descriptors", which it uses not only for outgoing messages, but also to hold state for bulk transfers requested by incoming messages. This pool should scale with the total number of peers.

To enable the building of the Portals LND (ptllnd.ko) configure with this option:

```
./configure --with-portals=<path-to-portals-headers>
```

Variable	Description
ntx (256)	Total number of messaging descriptors.
concurrent_peers (1152)	Maximum number of concurrent peers. Peers that attempt to connect beyond the maximum are not allowed.
peer_hash_table_size (101)	Number of hash table slots for the peers. This number should scale with concurrent_peers. The size of the peer hash table is set by the module parameter peer_hash_table_size which defaults to a value of 101. This number should be prime to ensure the peer hash table is populated evenly. It is advisable to increase this value to 1001 for ~10000 peers.
cksum (0)	Set to non-zero to enable message (not RDMA) checksums for outgoing packets. Incoming packets are always check-summed if necessary, independent of this value.
timeout (50)	Amount of time (in seconds) that a request can linger in a peers-active queue before the peer is considered dead.
portal (9)	Portal ID to use for the ptllnd traffic.
rxb_npages (64 * #cpus)	Number of pages in an RX buffer.

Variable	Description
credits (128)	Maximum total number of concurrent sends that are outstanding to a single peer at a given time.
peercredits (8)	Maximum number of concurrent sends that are outstanding to a single peer at a given time.
max_msg_size (512)	Maximum immediate message size. This MUST be the same on all nodes in a cluster. A peer that connects with a different max_msg_size value will be rejected.

30.2.8 Portals LND (Catamount)

The Portals LND Catamount (ptllnd) can be used as a interface layer to communicate with Sandia Portals networking devices. This version is intended to work on the Cray XT3 Catamount nodes using Cray Portals as a network transport.

To enable the building of the Portals LND configure with this option:

```
./configure --with-portals=<path-to-portals-headers>
```

The following PTLLND tunables are currently available:

Variable	Description
PTLLND_DEBUG (boolean, dflt 0)	Enables or disables debug features.
PTLLND_TX_HISTORY (int, dflt debug?1024:0)	Sets the size of the history buffer.
PTLLND_ABORT_ON_PROTOCOL_MISMATCH (boolean, dflt 1)	Calls abort action on connecting to a peer running a different version of the ptllnd protocol.
PTLLND_ABORT_ON_NAK (boolean, dflt 0)	Calls abort action when a peer sends a NAK. (Example: When it has timed out this node.)
PTLLND_DUMP_ON_NAK (boolean, dflt debug?1:0)	Dumps peer debug and the history on receiving a NAK.

Variable	Description
PTLLND_WATCHDOG_INTERVAL (int, dflt 1)	Sets intervals to check some peers for timed out communications while the application blocks for communications to complete.
PTLLND_TIMEOUT (int, dflt 50)	The communications timeout (in seconds).
PTLLND_LONG_WAIT (int, dflt debug?5:PTLLND_TIMEOUT)	The time (in seconds) after which the ptllnd prints a warning if it blocks for a longer time during connection establishment, cleanup after an error, or cleanup during shutdown.

The following environment variables can be set to configure the PTLLND's behavior.

Variable	Description
PTLLND_PORTAL (9)	The portal ID (PID) to use for the ptllnd traffic.
PTLLND_PID (9)	The virtual PID on which to contact servers.
PTLLND_PEERCREDITS (8)	The maximum number of concurrent sends that are outstanding to a single peer at any given instant.
PTLLND_MAX_MESSAGE_SIZE (512)	The maximum messages size. This MUST be the same on all nodes in a cluster.
PTLLND_MAX_MSGS_PER_BUFFER (64)	The number of messages in a receive buffer. Receive buffer will be allocated of size PTLLND_MAX_MSGS_PER_BUFFER times PTLLND_MAX_MESSAGE_SIZE.
PTLLND_MSG_SPARE (256)	Additional receive buffers posted to portals.
PTLLND_PEER_HASH_SIZE (101)	Number of hash table slots for the peers.
PTLLND_EQ_SIZE (1024)	Size of the Portals event queue (that is, maximum number of events in the queue).

30.2.9 MX LND

MXLND supports a number of load-time parameters using Linux's module parameter system. The following variables are available:

Variable	Description
n_waitd	Number of completion daemons.
max_peers	Maximum number of peers that may connect.
cksum	Enables small message (< 4 KB) checksums if set to a non-zero value.
ntx	Number of total tx message descriptors.
credits	Number of concurrent sends to a single peer.
board	Index value of the Myrinet board (NIC).
ep_id	MX endpoint ID.
polling	Use zero (0) to block (wait). A value > 0 will poll that many times before blocking.
hosts	IP-to-hostname resolution file.

Of the described variables, only hosts is required. It must be the absolute path to the MXLND hosts file.

For example:

```
options kmxlnd hosts=/etc/hosts.mxlnd
```

The file format for the hosts file is:

```
IP  HOST  BOARD   EP_ID
```

The values must be space and/or tab separated where:

`IP` is a valid IPv4 address

`HOST` is the name returned by `hostname` on that machine

`BOARD` is the index of the Myricom NIC (0 for the first card, etc.)

`EP_ID` is the MX endpoint ID

To obtain the optimal performance for your platform, you may want to vary the remaining options.

`n_waitd` (1) sets the number of threads that process completed MX requests (sends and receives).

`max_peers` (1024) tells MXLND the upper limit of machines that it will need to communicate with. This affects how many receives it will pre-post and each receive will use one page of memory. Ideally, on clients, this value will be equal to the total number of Lustre servers (MDS and OSS). On servers, it needs to equal the total number of machines in the storage system. cksum (0) turns on small message checksums. It can be used to aid in troubleshooting. MX also provides an optional checksumming feature which can check all messages (large and small). For details, see the MX README.

`ntx` (256) is the number of total sends in flight from this machine. In actuality, MXLND reserves half of them for connect messages so make this value twice as large as you want for the total number of sends in flight.

`credits` (8) is the number of in-flight messages for a specific peer. This is part of the flow-control system in Lustre. Increasing this value may improve performance but it requires more memory because each message requires at least one page.

`board` (0) is the index of the Myricom NIC. Hosts can have multiple Myricom NICs and this identifies which one MXLND should use. This value must match the board value in your MXLND hosts file for this host.

`ep_id` (3) is the MX endpoint ID. Each process that uses MX is required to have at least one MX endpoint to access the MX library and NIC. The ID is a simple index starting at zero (0). This value must match the endpoint ID value in your MXLND hosts file for this host.

`polling` (0) determines whether this host will poll or block for MX request completions. A value of 0 blocks and any positive value will poll that many times before blocking. Since polling increases CPU usage, we suggest that you set this to zero (0) on the client and experiment with different values for servers.

CHAPTER 31

System Configuration Utilities (man8)

This chapter includes system configuration utilities and includes the following sections:

31.1 mkfs.lustre

The mkfs.lustre utility formats a disk for a Lustre service.

Synopsis

```
mkfs.lustre <target_type> [options] device
```

where <*target_type*> is one of the following:

Option	Description
--ost	Object Storage Target (OST)
--mdt	Metadata Storage Target (MDT)
--mgs	Configuration Management Service (MGS), one per site. This service can be combined with one **--mdt** service by specifying both types.

Description

`mkfs.lustre` is used to format a disk device for use as part of a Lustre file system. After formatting, a disk can be mounted to start the Lustre service defined by this command.

When the file system is created, parameters can simply be added as a `--param` option to the `mkfs.lustre` command. See Setting Parameters with mkfs.lustre.

Option	Description
--backfstype=*fstype*	Forces a particular format for the backing file system (such as ext3, ldiskfs).
--comment=*comment*	Sets a user comment about this disk, ignored by Lustre.
--device-size=*KB*	Sets the device size for loop and non-loop devices.

Option	Description
--dryrun	Only prints what would be done; it does not affect the disk.
--failnode=*nid,...*	Sets the NID(s) of a failover partner. This option can be repeated as needed.
--fsname=*filesystem_name*	The Lustre file system of which this service/node will be a part. The default file system name is “lustre”. NOTE: The file system name is limited to 8 characters.
--index=*index*	Forces a particular OST or MDT index.
--mkfsoptions=*opts*	Formats options for the backing file system. For example, ext3 options could be set here.
--mountfsoptions=*opts*	Sets permanent mount options. This is equivalent to the setting in /etc/fstab.
--mgsnode=*nid,...*	Sets the NIDs of the MGS node, required for all targets other than the MGS.
--param key=value	Sets the permanent parameter key to value. This option can be repeated as desired. Typical options might include: **--param sys.timeout=40** System obd timeout. **--param lov.stripesize=2M** Default stripe size. **--param lov.stripecount=2** Default stripe count. **--param failover.mode=failout** Returns errors instead of waiting for recovery.
--quiet	Prints less information.

Option	Description
--reformat	Reformats an existing Lustre disk.
--stripe-count-hint=*stripes*	Used to optimize the MDT's inode size.
--verbose	Prints more information.

Examples

Creates a combined MGS and MDT for file system **testfs** on node **cfs21**:

```
mkfs.lustre --fsname=testfs --mdt --mgs /dev/sda1
```

Creates an OST for file system **testfs** on any node (using the above MGS):

```
mkfs.lustre --fsname=testfs --ost --mgsnode=cfs21@tcp0 /dev/sdb
```

Creates a standalone MGS on, e.g., node **cfs22**:

```
mkfs.lustre --mgs /dev/sda1
```

Creates an MDT for file system **myfs1** on any node (using the above MGS):

```
mkfs.lustre --fsname=myfs1 --mdt --mgsnode=cfs22@tcp0 /dev/sda2
```

31.2 tunefs.lustre

The tunefs.lustre utility modifies configuration information on a Lustre target disk.

Synopsis

```
tunefs.lustre [options] device
```

Description

`tunefs.lustre` is used to modify configuration information on a Lustre target disk. This includes upgrading old (pre-Lustre 1.8) disks. This does not reformat the disk or erase the target information, but modifying the configuration information can result in an unusable file system.

Caution – Changes made here affect a file system when the target is mounted the next time.

With `tunefs.lustre`, parameters are "additive" -- new parameters are specified in addition to old parameters, they do not replace them. To erase all old `tunefs.lustre` parameters and just use newly-specified parameters, run:

```
$ tunefs.lustre --erase-params --param=<new parameters>
```

The `tunefs.lustre` command can be used to set any parameter settable in a /proc/fs/lustre file and that has its own OBD device, so it can be specified as `<obd|fsname>.<obdtype>.<proc_file_name>=<value>`. For example:

```
$ tunefs.lustre --param mdt.group_upcall=NONE /dev/sda1
```

Options

The `tunefs.lustre` options are listed and explained below.

Option	Description

--comment=*comment*

Sets a user comment about this disk, ignored by Lustre.

--dryrun

Only prints what would be done; does not affect the disk.

--erase-params

Removes all previous parameter information.

--failnode=*nid,...*

Sets the NID(s) of a failover partner. This option can be repeated as needed.

--fsname=*filesystem_name*

The Lustre file system of which this service will be a part. The default file system name is “lustre”.

--index=*index*

Forces a particular OST or MDT index.

--mountfsoptions=*opts*

Sets permanent mount options; equivalent to the setting in /etc/fstab.

--mgs

Adds a configuration management service to this target.

--msgnode=*nid,...*

Sets the NID(s) of the MGS node; required for all targets other than the MGS.

--nomgs

Removes a configuration management service to this target.

--quiet

Prints less information.

Option	Description
--verbose	Prints more information.
--writeconf	Erases all configuration logs for the file system to which this MDT belongs, and regenerates them. This is very dangerous. All clients and servers should be stopped. All targets must then be restarted to regenerate the logs. No clients should be started until all targets have restarted. In general, this command should only be executed on the MDT, not the OSTs.

Examples

Changing the MGS's NID address. (This should be done on each target disk, since they should all contact the same MGS.)

```
tunefs.lustre --erase-param --mgsnode=<new_nid> --writeconf /dev/sda
```

Adding a failover NID location for this target.

```
tunefs.lustre --param="failover.node=192.168.0.13@tcp0" /dev/sda
```

31.3 lctl

The `lctl` utility is used for root control and configuration. With `lctl` you can directly control Lustre via an ioctl interface, allowing various configuration, maintenance and debugging features to be accessed.

Synopsis

```
lctl
lctl --device <OST device number> <command [args]>
```

Description

The lctl utility can be invoked in interactive mode by issuing the lctl command. After that, commands are issued as shown below. The most common lctl commands are:

```
dl
device
network <up/down>
list_nids
ping {nid}
help
quit
```

For a complete list of available commands, type `help` at the `lctl` prompt. To get basic help on command meaning and syntax, type `help` *`command`*

For non-interactive use, use the second invocation, which runs the command after connecting to the device.

Setting Parameters with lctl

Lustre parameters are not always accessible using the procfs interface, as it is platform-specific. As a solution, `lctl {get,set}_param` has been introduced as a platform-independent interface to the Lustre tunables. Avoid direct references to /proc/{fs,sys}/{lustre,lnet}. For future portability, use `lctl {get,set}_param` .

When the file system is running, temporary parameters can be set using the `lctl set_param` command. These parameters map to items in `/proc/{fs,sys}/{lnet,lustre}`. The `lctl set_param` command uses this syntax:

```
lctl set_param [-n] <obdtype>.<obdname>.<proc_file_name>=<value>
```

For example:

```
$ lctl set_param ldlm.namespaces.*osc*.lru_size=$((NR_CPU*100))
```

Many permanent parameters can be set with the `lctl conf_param` command. In general, the `lctl conf_param` command can be used to specify any parameter settable in a /proc/fs/lustre file, with its own OBD device. The `lctl conf_param` command uses this syntax:

```
<obd|fsname>.<obdtype>.<proc_file_name>=<value>)
```

For example:

```
$ lctl conf_param testfs-MDT0000.mdt.group_upcall=NONE
$ lctl conf_param testfs.llite.max_read_ahead_mb=16
```

Caution – The `lctl conf_param` command permanently sets parameters in the file system configuration.

To get current Lustre parameter settings, use the `lctl get_param` command with this syntax:

```
lctl get_param [-n] <obdtype>.<obdname>.<proc_file_name>
```

For example:

```
$ lctl get_param -n ost.*.ost_io.timeouts
```

To list Lustre parameters that are available to set, use the `lctl list_param` command, with this syntax:

```
lctl list_param [-n] <obdtype>.<obdname>
```

For example:

```
$ lctl list_param obdfilter.lustre-OST0000
```

Network Configuration

Option	Description
network <up/down>|<tcp/elan/myrinet>	Starts or stops LNET. Or, select a network type for other lctl LNET commands.
list_nids	Prints all NIDs on the local node. LNET must be running.
which_nid <nidlist>	From a list of NIDs for a remote node, identifies the NID on which interface communication will occur.
ping {nid}	Check's LNET connectivity via an LNET ping. This uses the fabric appropriate to the specified NID.
interface_list	Prints the network interface information for a given network type.
peer_list	Prints the known peers for a given network type.
conn_list	Prints all the connected remote NIDs for a given network type.
active_tx	This command prints active transmits. It is only used for the Elan network type.

Device Operations

Option	Description
lctl get_param [-n] <path_name>	Gets the Lustre or LNET parameters from the specified <path_name>. Use the **-n** option to get only the parameter value and skip the pathname in the output. **NOTE**: Lustre tunables are not always accessible using procfs interface, as it is platform-specific. As a solution, **lctl {get,set}_param** has been introduced as a platform-independent interface to the Lustre tunables. Avoid direct references to **/proc/{fs,sys}/{lustre,lnet}**. For future portability, use **lctl {get,set}_param** instead.
lctl set_param [-n] <path_name>	Sets the specified value to the Lustre or LNET parameter indicated by the pathname. Use the **-n** option to skip the pathname in the output. **NOTE**: Lustre tunables are not always accessible using procfs interface, as it is platform-specific. As a solution, **lctl {get,set}_param** has been introduced as a platform-independent interface to the Lustre tunables. Avoid direct references to **/proc/{fs,sys}/{lustre,lnet}**. For future portability, use **lctl {get,set}_param** instead.
conf_param <device> <parameter>	Sets a permanent configuration parameter for any device via the MGS. This command must be run on the MGS node.
activate	Re-activates an import after the de-activate operation.
deactivate	Running `lctl deactivate` on the MDS stops new objects from being allocated on the OST. Running `lctl deactivate` on Lustre clients causes them to return -EIO when accessing objects on the OST instead of waiting for recovery.
abort_recovery	Aborts the recovery process on a re-starting MDT or OST device.

Virtual Block Device Operations

Lustre can emulate a virtual block device upon a regular file. This emulation is needed when you are trying to set up a swap space via the file.

Option	Description
blockdev_attach <file name> <device node>	Attaches a regular Lustre file to a block device. If the device node is non-existent, lctl creates it. We recommend that you create the device node by lctl since the emulator uses a dynamic major number.
blockdev_detach <device node>	Detaches the virtual block device.
blockdev_info <device node>	Provides information on which Lustre file is attached to the device node.

Debug

Option	Description
debug_daemon	Starts and stops the debug daemon, and controls the output filename and size.
debug_kernel [file] [raw]	Dumps the kernel debug buffer to stdout or a file.
debug_file <input> [output]	Converts the kernel-dumped debug log from binary to plain text format.
clear	Clears the kernel debug buffer.
mark <text>	Inserts marker text in the kernel debug buffer.

Options

Use the following options to invoke `lctl`.

Option	Description
--device	Device to be used for the operation (specified by name or number). See **device_list**.
--ignore_errors | ignore_errors	Ignores errors during script processing.

Examples

lctl

```
$ lctl
lctl > dl

    0    UP    mgc    MGC192.168.0.20@tcp    bfbb24e3-7deb-2ffa-
eab0-44dffe00f692 5
    1 UP ost OSS OSS_uuid 3
    2 UP obdfilter testfs-OST0000 testfs-OST0000_UUID 3
lctl > dk /tmp/log Debug log: 87 lines, 87 kept, 0 dropped.
lctl > quit

$ lctl conf_param testfs-MDT0000 sys.timeout=40
$ lctl conf_param testfs-MDT0000.lov.stripesize=2M
$ lctl conf_param testfs-OST0000.osc.max_dirty_mb=29.15
$ lctl conf_param testfs-OST0000.ost.client_cache_seconds=15
```

get_param

```
$ lctl
lctl > get_param obdfilter.lustre-OST0000.kbytesavail
obdfilter.lustre-OST0000.kbytesavail=249364
lctl > get_param -n obdfilter.lustre-OST0000.kbytesavail
249364
lctl > get_param timeout
timeout=20
lctl > get_param -n timeout
20
lctl > get_param obdfilter.*.kbytesavail
obdfilter.lustre-OST0000.kbytesavail=249364
obdfilter.lustre-OST0001.kbytesavail=249364
lctl >
```

set_param

```
$ lctl > set_param obdfilter.*.kbytesavail=0
obdfilter.lustre-OST0000.kbytesavail=0
obdfilter.lustre-OST0001.kbytesavail=0
lctl > set_param -n obdfilter.*.kbytesavail=0
lctl > set_param fail_loc=0
fail_loc=0
```

31.4 mount.lustre

The `mount.lustre` utility starts a Lustre client or target service.

Synopsis

```
mount -t lustre [-o options] directory
```

Description

The `mount.lustre` utility starts a Lustre client or target service. This program should not be called directly; rather, it is a helper program invoked through `mount(8)`, as shown above. Use the `umount(8)` command to stop Lustre clients and targets.

There are two forms for the device option, depending on whether a client or a target service is started:

Option	Description
<mgsspec>:/<fsname>	This mounts the Lustre file system, <fsname>, by contacting the Management Service at <mgsspec> on the pathname given by <directory>. The format for <mgsspec> is defined below. A mounted file system appears in fstab(5) and is usable, like any local file system, providing a full POSIX-compliant interface.
<disk_device>	This starts the target service defined by the `mkfs.lustre` command on the physical disk <disk_device>. A mounted target service file system is only useful for df(1) operations and appears in fstab(5) to show the device is in use.

Options

Option	Description
<mgsspec>:=<mgsnode>[:<mgsnode>]	The MGS specification may be a colon-separated list of nodes.
<mgsnode>:=<mgsnid>[,<mgsnid>]	Each node may be specified by a comma-separated list of NIDs.

In addition to the standard mount options, Lustre understands the following client-specific options:

Option	Description
flock	Enables flock support, coherent across client nodes.
localflock	Enables local flock support, using only client-local flock (faster, for applications that require flock, but do not run on multiple nodes).
noflock	Disables flock support entirely. Applications calling flock get an error.
user_xattr	Enables get/set of extended attributes by regular users.
nouser_xattr	Disables use of extended attributes by regular users. Root and system processes can still use extended attributes.
acl	Enables ACL support.
noacl	Disables ACL support.

In addition to the standard mount options and backing disk type (e.g. ext3) options, Lustre understands the following server-specific options:

Option	Description
nosvc	Starts only the MGC (and MGS, if co-located) for a target service, not the actual service.
nomgs	Starts only the MDT (with a co-located MGS), without starting the MGS.
exclude=<ostlist>	Starts a client or MDT with a colon-separated list of known inactive OSTs.
abort_recov	Aborts client recovery and immediately starts the target service.
md_stripe_cache_size	Sets the stripe cache size for server-side disk with a striped RAID configuration.
recovery_time_soft=<timeout>	Allows <timeout> seconds for clients to reconnect for recovery after a server crash. This timeout is incrementally extended if it is about to expire and the server is still handling new connections from recoverable clients. The default soft recovery timeout is 300 seconds (5 minutes).
recovery_time_hard=<timeout>	The server is allowed to incrementally extend its timeout, up to a hard maximum of <timeout> seconds. The default hard recovery timeout is 900 seconds (15 minutes).

Examples

Starts a client for the Lustre file system `testfs` at mount point `/mnt/myfilesystem`. The Management Service is running on a node reachable from this client via the `cfs21@tcp0` NID.

```
mount -t lustre cfs21@tcp0:/testfs /mnt/myfilesystem
```

Starts the Lustre target service on `/dev/sda1`.

```
mount -t lustre /dev/sda1 /mnt/test/mdt
```

Starts the `testfs-MDT0000` service (using the disk label), but aborts the recovery process.

```
mount -t lustre -L testfs-MDT0000 -o abort_recov /mnt/test/mdt
```

Note – If the Service Tags tool (from the sun-servicetag package) can be found in /opt/sun/servicetag/bin/stclient, an inventory service tag is created reflecting the Lustre service being provided. If this tool cannot be found, mount.lustre silently ignores it and no service tag is created. The stclient(1) tool only creates the local service tag. No information is sent to the asset management system until you run the Registration Client to collect the tags and then upload them to the inventory system using your inventory system account. For more information, see Service Tags.

31.5 Additional System Configuration Utilities

This section describes additional system configuration utilities that were added in Lustre 1.6.

31.5.1 lustre_rmmod.sh

The lustre_rmmod.sh utility removes all Lustre and LNET modules (assuming no Lustre services are running). It is located in `/usr/bin`.

Note – The `lustre_rmmod.sh` utility does not work if Lustre modules are being used or if you have manually run the `lctl network up` command.

31.5.2 e2scan

The e2scan utility is an ext2 file system-modified inode scan program. The e2scan program uses libext2fs to find inodes with `ctime` or `mtime` newer than a given time and prints out their pathname. Use e2scan to efficiently generate lists of files that have been modified. The e2scan tool is included in e2fsprogs, located at:

http://downloads.clusterfs.com/public/tools/e2fsprogs/latest

Synopsis

```
e2scan [options] [-f file] block_device
```

Description

When invoked, the e2scan utility iterates all inodes on the block device, finds modified inodes, and prints their inode numbers. A similar iterator, using libext2fs(5), builds a table (called parent database) which lists the parent node for each inode. With a lookup function, you can reconstruct modified pathnames from root.

Options

Option	Description
-b inode buffer blocks	Sets the readahead inode blocks to get excellent performance when scanning the block device.
-o output file	If an output file is specified, modified pathnames are written to this file. Otherwise, modified parameters are written to stdout.
-t inode | pathname	Sets the e2scan type if type is inode. The e2scan utility prints modified inode numbers to stdout. By default, the type is set as pathname. The e2scan utility lists modified pathnames based on modified inode numbers.
-u	Rebuilds the parent database from scratch. Otherwise, the current parent database is used.

31.5.3 Utilities to Manage Large Clusters

The following utilities are located in `/usr/bin`.

lustre_config.sh

The lustre_config.sh utility helps automate the formatting and setup of disks on multiple nodes. An entire installation is described in a comma-separated file and passed to this script, which then formats the drives, updates modprobe.conf and produces high-availability (HA) configuration files.

lustre_createcsv.sh

The lustre_createcsv.sh utility generates a CSV file describing the currently-running installation.

lustre_up14.sh

The lustre_up14.sh utility grabs client configuration files from old MDTs. When upgrading Lustre from 1.4.x to 1.6.x, if the MGS is not co-located with the MDT or the client name is non-standard, this utility is used to retrieve the old client log. For more information, see Upgrading and Downgrading Lustre.

31.5.4 Application Profiling Utilities

The following utilities are located in `/usr/bin`.

lustre_req_history.sh

The `lustre_req_history.sh` utility (run from a client), assembles as much Lustre RPC request history as possible from the local node and from the servers that were contacted, providing a better picture of the coordinated network activity.

llstat.sh

The `llstat.sh` utility (improved in Lustre 1.6), handles a wider range of `/proc` files, and has command line switches to produce more graphable output.

plot-llstat.sh

The `plot-llstat.sh` utility plots the output from `llstat.sh` using gnuplot.

31.5.5 More /proc Statistics for Application Profiling

The following utilities provide additional statistics.

vfs_ops_stats

The client `vfs_ops_stats` utility tracks Linux VFS operation calls into Lustre for a single PID, PPID, GID or everything.

```
/proc/fs/lustre/llite/*/vfs_ops_stats
/proc/fs/lustre/llite/*/vfs_track_[pid|ppid|gid]
```

extents_stats

The client `extents_stats` utility shows the size distribution of I/O calls from the client (cumulative and by process).

```
/proc/fs/lustre/llite/*/extents_stats, extents_stats_per_process
```

offset_stats

The client `offset_stats` utility shows the read/write seek activity of a client by offsets and ranges.

```
/proc/fs/lustre/llite/*/offset_stats
```

Lustre 1.6 included per-client and improved MDT statistics:

- Per-client statistics tracked on the servers

  Each MDT and OST now tracks LDLM and operations statistics for every connected client, for comparisons and simpler collection of distributed job statistics.

  ```
  /proc/fs/lustre/mds|obdfilter/*/exports/
  ```

- Improved MDT statistics

  More detailed MDT operations statistics are collected for better profiling.

  ```
  /proc/fs/lustre/mds/*/stats
  ```

31.5.6 Testing / Debugging Utilities

The following utilities are located in `/usr/bin`.

loadgen

The `loadgen` utility is a test program you can use to generate large loads on local or remote OSTs or echo servers. For more information on loadgen and its usage, refer to:

```
https://mail.clusterfs.com/wikis/lustre/LoadGen
```

llog_reader

The `llog_reader` utility translates a Lustre configuration log into human-readable form.

lr_reader

The `lr_reader` utility translates a last received (`last_rcvd`) file into human-readable form.

31.5.7 Flock Feature

Lustre now includes the flock feature, which provides file locking support. Flock describes classes of file locks known as 'flocks'. Flock can apply or remove a lock on an open file as specified by the user. However, a single file may not, simultaneously, have both shared and exclusive locks.

By default, the flock utility is disabled on Lustre. Two modes are available.

local mode In this mode, locks are coherent on one node (a single-node flock), but not across all clients. To enable it, use `-o localflock`.
This is a client-mount option.

NOTE: This mode does not impact performance and is appropriate for single-node databases.

consistent mode In this mode, locks are coherent across all clients.
To enable it, use the `-o flock`. This is a client-mount option.

CAUTION: This mode has a noticeable performance impact and may affect stability, depending on the Lustre version used. Consider using a newer Lustre version which is more stable.

A call to use flock may be blocked if another process is holding an incompatible lock. Locks created using flock are applicable for an open file table entry. Therefore, a single process may hold only one type of lock (shared or exclusive) on a single file. Subsequent flock calls on a file that is already locked converts the existing lock to the new lock mode.

31.5.7.1 Example

```
$ mount -t lustre –o flock mds@tcp0:/lustre /mnt/client
```

You can check it in `/etc/mtab`. It should look like,

```
mds@tcp0:/lustre /mnt/client lustre rw,flock 00
```

31.5.8 l_getgroups

The l_getgroups utility handles Lustre user / group cache upcall.

Synopsis

```
l_getgroups [-v] [-d | mdsname] uid
l_getgroups [-v] -s
```

Options

Option	Description
--d	
	Debug - prints values to stdout instead of Lustre.
-s	
	Sleep - mlock memory in core and sleeps forever.
-v	
	Verbose - Logs start/stop to syslog.
mdsname	
	MDS device name.

Description

The group upcall file contains the path to an executable file that, when properly installed, is invoked to resolve a numeric UID to a group membership list. This utility should complete the mds_grp_downcall_data structure and write it to the `/proc/fs/lustre/mds/mds service/group_info` pseudo-file.

The l_getgroups utility is the reference implementation of the user or group cache upcall.

Files

The l_getgroups files are located at:

`/proc/fs/lustre/mds/mds-service/group_upcall`

31.5.9 llobdstat

The llobdstat utility displays OST statistics.

Synopsis

```
llobdstat ost_name [interval]
```

Description

The llobdstat utility displays a line of OST statistics for a given OST at specified intervals (in seconds).

Option	Description
ost_name	Name of the OBD for which statistics are requested.
interval	Time interval (in seconds) after which statistics are refreshed.

Example

```
# llobdstat liane-OST0002 1
/usr/bin/llobdstat on /proc/fs/lustre/obdfilter/liane-OST0002/stats
Processor counters run at 2800.189 MHz
Read: 1.21431e+07, Write: 9.93363e+08, create/destroy: 24/1499, stat: 34,
punch: 18
[NOTE: cx: create, dx: destroy, st: statfs, pu: punch ]
Timestamp   Read-delta  ReadRate  Write-delta  WriteRate
--------------------------------------------------------
1217026053    0.00MB    0.00MB/s     0.00MB    0.00MB/s
1217026054    0.00MB    0.00MB/s     0.00MB    0.00MB/s
1217026055    0.00MB    0.00MB/s     0.00MB    0.00MB/s
1217026056    0.00MB    0.00MB/s     0.00MB    0.00MB/s
1217026057    0.00MB    0.00MB/s     0.00MB    0.00MB/s
1217026058    0.00MB    0.00MB/s     0.00MB    0.00MB/s
1217026059    0.00MB    0.00MB/s     0.00MB    0.00MB/s st:1
```

Files

The llobdstat files are located at:

```
/proc/fs/lustre/obdfilter/<ostname>/stats
```

31.5.10 llstat

The llstat utility displays Lustre statistics.

Synopsis

```
llstat [-c] [-g] [-i interval] stats_file
```

Description

The llstat utility displays statistics from any of the Lustre statistics files that share a common format and are updated at a specified interval (in seconds). To stop statistics printing, type CTRL-C.h

Options

Option	Description
-c	Clears the statistics file.
-i	Specifies the interval polling period (in seconds).
-g	Specifies graphable output format.
-h	Displays help information.
stats_file	Specifies either the full path to a statistics file or a shorthand reference, **mds** or **ost**

Example

To monitor `/proc/fs/lustre/ost/OSS/ost/stats` at 1 second intervals, run;

```
llstat -i 1 ost
```

Files

The llstat files are located at:

```
/proc/fs/lustre/mdt/MDS/*/stats
/proc/fs/lustre/mds/*/exports/*/stats
/proc/fs/lustre/mdc/*/stats
/proc/fs/lustre/ldlm/services/*/stats
/proc/fs/lustre/ldlm/namespaces/*/pool/stats
/proc/fs/lustre/mgs/MGS/exports/*/stats
/proc/fs/lustre/ost/OSS/*/stats
/proc/fs/lustre/osc/*/stats
/proc/fs/lustre/obdfilter/*/exports/*/stats
/proc/fs/lustre/obdfilter/*/stats
/proc/fs/lustre/llite/*/stats
```

31.5.11 lst

The lst utility starts LNET self-test.

Synopsis

```
lst
```

Description

LNET self-test helps site administrators confirm that Lustre Networking (LNET) has been correctly installed and configured. The self-test also confirms that LNET, the network software and the underlying hardware are performing as expected.

Each LNET self-test runs in the context of a session. A node can be associated with only one session at a time, to ensure that the session has exclusive use of the nodes on which it is running. A single node creates, controls and monitors a single session. This node is referred to as the self-test console.

Any node may act as the self-test console. Nodes are named and allocated to a self-test session in groups. This allows all nodes in a group to be referenced by a single name.

Test configurations are built by describing and running test batches. A test batch is a named collection of tests, with each test composed of a number of individual point-to-point tests running in parallel. These individual point-to-point tests are instantiated according to the test type, source group, target group and distribution specified when the test is added to the test batch.

Modules

To run LNET self-test, load following modules: libcfs, lnet, lnet_selftest and any one of the klnds (ksocklnd, ko2iblnd...). To load all necessary modules, run modprobe lnet_selftest, which recursively loads the modules on which lnet_selftest depends.

There are two types of nodes for LNET self-test: console and test. Both node types require all previously-specified modules to be loaded. (The userspace test node does not require these modules).

Test nodes can either be in kernel or in userspace. A console user can invite a kernel test node to join the test session by running lst add_group NID, but the user cannot actively add a userspace test node to the test-session. However, the console user can passively accept a test node to the test session while the test node runs lst client to connect to the console.

Utilities

LNET self-test includes two user utilities, lst and lstclient.

lst is the user interface for the self-test console (run on console node). It provides a list of commands to control the entire test system, such as create session, create test groups, etc.

lstclient is the userspace self-test program which is linked with userspace LNDs and LNET. A user can invoke lstclient to join a self-test session:

```
lstclient -sesid CONSOLE_NID group NAME
```

Example

This is an example of an LNET self-test script which simulates the traffic pattern of a set of Lustre servers on a TCP network, accessed by Lustre clients on an IB network (connected via LNET routers), with half the clients reading and half the clients writing.

```
#!/bin/bash
export LST_SESSION=$$
lst new_session read/write
lst add_group servers 192.168.10.[8,10,12-16]@tcp
lst add_group readers 192.168.1.[1-253/2]@o2ib
lst add_group writers 192.168.1.[2-254/2]@o2ib
lst add_batch bulk_rw
lst add_test --batch bulk_rw --from readers --to servers    brw read
check=simple size=1M
lst add_test --batch bulk_rw --from writers --to servers    brw write
check=full size=4K
# start running
lst run bulk_rw
# display server stats for 30 seconds
lst stat servers & sleep 30; kill $!
# tear down
lst end_session
```

31.5.12 plot-llstat

The plot-llstat utility plots Lustre statistics.

Synopsis

```
plot-llstat results_filename [parameter_index]
```

Options

Option	Description
results_filename	
	Output generated by plot-llstat
parameter_index	
	Value of parameter_index can be: 1 - count per interval 2 - count per second (default setting) 3 - total count

Description

The plot-llstat utility generates a CSV file and instruction files for gnuplot from llstat output. Since llstat is generic in nature, plot-llstat is also a generic script. The value of parameter_index can be 1 for count per interval, 2 for count per second (default setting) or 3 for total count.

The plot-llstat utility creates a .dat (CSV) file using the number of operations specified by the user. The number of operations equals the number of columns in the CSV file. The values in those columns are equal to the corresponding value of parameter_index in the output file.

The plot-llstat utility also creates a .scr file that contains instructions for gnuplot to plot the graph. After generating the .dat and .scr files, the plot llstat tool invokes gnuplot to display the graph.

Example

```
llstat -i2 -g -c lustre-OST0000 > log
plot-llstat log 3
```

31.5.13 routerstat

The routerstat utility prints Lustre router statistics.

Synopsis

```
routerstat [interval]
```

Description

The routerstat utility watches LNET router statistics. If no interval is specified, then statistics are sampled and printed only one time. Otherwise, statistics are sampled and printed at the specified interval (in seconds).

Options

The routerstat output includes the following fields:

Field	Description
M	msgs_alloc(msgs_max)
E	errors
S	send_length/send_count
R	recv_length/recv_count
F	route_length/route_count
D	drop_length/drop_count

Files

Routerstat extracts statistics data from:

```
/proc/sys/lnet/stats
```

31.5.14 ll_recover_lost_found_objs

The ll_recover_lost_found_objs utility helps recover Lustre OST objects (file data) from a lost and found directory back to their correct locations.

Running the ll_recover_lost_found_objs tool is not strictly necessary to bring an OST back online, it just avoids losing access to objects that were moved to the lost and found directory due to directory corruption.

Synopsis

```
$ ll_recover_lost_found_objs [-hv] -d directory
```

Description

The first time Lustre writes to an object, it saves the MDS inode number and the objid as an extended attribute on the object, so in case of directory corruption of the OST, it is possible to recover the objects. Running e2fsck fixes the corrupted OST directory, but it puts all of the objects into a lost and found directory, where they are inaccessible to Lustre. Use the `ll_recover_lost_found_objs` utility to recover all (or at least most) objects from a lost and found directory back to their place in the O/0/d* directories.

To use ll_recover_lost_found_objs, mount the file system locally (using the -t ldiskfs command), run the utility and then unmount it again. The OST must not be mounted by Lustre when ll_recover_lost_found_objs is run.

Options

Field	Description
-h	Prints a help message
-v	Increases verbosity
-d directory	Sets the lost and found directory path

Example

```
ll_recover_lost_found_objs -d /mnt/ost/lost+found
```

CHAPTER 32

System Limits

This chapter describes various limits on the size of files and file systems. These limits are imposed by either the Lustre architecture or the Linux VFS and VM subsystems. In a few cases, a limit is defined within the code and could be changed by re-compiling Lustre. In those cases, the selected limit is supported by Lustre testing and may change in future releases. This chapter includes the following sections:

32.1 Maximum Stripe Count

The maximum number of stripe count is 160. This limit is hard-coded, but is near the upper limit imposed by the underlying ext3 file system. It may be increased in future releases. Under normal circumstances, the stripe count is not affected by ACLs.

32.2 Maximum Stripe Size

For a 32-bit machine, the product of stripe size and stripe count (stripe_size * stripe_count) must be less than 2^32. The ext3 limit of 2TB for a single file applies for a 64-bit machine. (Lustre can support 160 stripes of 2 TB each on a 64-bit system.)

32.3 Minimum Stripe Size

Due to the 64 KB PAGE_SIZE on some 64-bit machines, the minimum stripe size is set to 64 KB.

32.4 Maximum Number of OSTs and MDTs

You can set the maximum number of OSTs by a compile option. The limit of 1020 OSTs in Lustre release 1.4.7 is increased to a maximum of 8150 OSTs in 1.6.0. Testing is in progress to move the limit to 4000 OSTs.

The maximum number of MDSs will be determined after accomplishing MDS clustering.

32.5 Maximum Number of Clients

Currently, the number of clients is limited to 131072. We have tested up to 22000 clients.

32.6 Maximum Size of a File System

For i386 systems with 2.6 kernels, the block devices are limited to 16 TB. Each OST or MDT can have a file system up to 16 TB, regardless of whether 32-bit or 64-bit kernels are on the server.

You can have multiple OST file systems on a single node. Currently, the largest production Lustre file system has 448 OSTs in a single file system. There is a compile-time limit of 8150 OSTs in a single file system, giving a theoretical file system limit of nearly 64 PB.

Several production Lustre file systems have around 200 OSTs in a single file system. The largest file system in production is at least 1.3 PB (184 OSTs). All these facts indicate that Lustre would scale just fine if more hardware is made available.

32.7 Maximum File Size

Individual files have a hard limit of nearly 16 TB on 32-bit systems imposed by the kernel memory subsystem. On 64-bit systems this limit does not exist. Hence, files can be 64-bits in size. Lustre imposes an additional size limit of up to the number of stripes, where each stripe is 2 TB. A single file can have a maximum of 160 stripes, which gives an upper single file limit of 320 TB for 64-bit systems. The actual amount of data that can be stored in a file depends upon the amount of free space in each OST on which the file is striped.

32.8 Maximum Number of Files or Subdirectories in a Single Directory

Lustre uses the ext3 hashed directory code, which has a limit of about 25 million files. On reaching this limit, the directory grows to more than 2 GB depending on the length of the filenames. The limit on subdirectories is the same as the limit on regular files in all later versions of Lustre due to a small ext3 format change.

In fact, Lustre is tested with ten million files in a single directory. On a properly-configured dual-CPU MDS with 4 GB RAM, random lookups in such a directory are possible at a rate of 5,000 files / second.

32.9 MDS Space Consumption

A single MDS imposes an upper limit of 4 billion inodes. The default limit is slightly less than the device size of 4 KB, meaning 512 MB inodes for a file system with MDS of 2 TB. This can be increased initially, at the time of MDS file system creation, by specifying the `--mkfsoptions='-i 2048'` option on the `--add mds` config line for the MDS.

For newer releases of e2fsprogs, you can specify `'-i 1024'` to create 1 inode for every 1 KB disk space. You can also specify `'-N {num inodes}'` to set a specific number of inodes. The inode size (-I) should not be larger than half the inode ratio (`-i`). Otherwise, mke2fs will spin trying to write more number of inodes than the inodes that can fit into the device.

For more information, see Options for Formatting the MDT and OSTs.

32.10 Maximum Length of a Filename and Pathname

This limit is 255 bytes for a single filename, the same as in an ext3 file system. The Linux VFS imposes a full pathname length of 4096 bytes.

32.11 Maximum Number of Open Files for Lustre File Systems

Lustre does not impose maximum number of open files, but practically it depends on amount of RAM on the MDS. There are no "tables" for open files on the MDS, as they are only linked in a list to a given client's export. Each client process probably has a limit of several thousands of open files which depends on the ulimit.

32.12 OSS RAM Size

For a single OST, there is no strict rule to size the OSS RAM. However, as a guideline for Lustre 1.8 installations, 2 GB per OST is a reasonable RAM size. For details on determining the memory needed for an OSS node, see OSS Memory Requirements

Glossary

A

ACL Access Control List - An extended attribute associated with a file which contains authorization directives.

Administrative OST failure A configuration directive given to a cluster to declare that an OST has failed, so errors can be immediately returned.

C

CFS Cluster File Systems, Inc., a United States corporation founded in 2001 by Peter J. Braam to develop, maintain and support Lustre.

CMD Clustered metadata, a collection of metadata targets implementing a single file system namespace.

Completion Callback An RPC made by an OST or MDT to another system, usually a client, to indicate that the lock request is now granted.

Configlog An llog file used in a node, or retrieved from a management server over the network with configuration instructions for Lustre systems at startup time.

Configuration Lock A lock held by every node in the cluster to control configuration changes. When callbacks are received, the nodes quiesce their traffic, cancel the lock and await configuration changes after which they reacquire the lock before resuming normal operation.

D

Default stripe pattern Information in the LOV descriptor that describes the default stripe count used for new files in a file system. This can be amended by using a directory stripe descriptor or a per-file stripe descriptor.

Direct I/O A mechanism which can be used during read and write system calls. It bypasses the kernel. I/O cache to memory copy of data between kernel and application memory address spaces.

Directory stripe descriptor An extended attribute that describes the default stripe pattern for files underneath that directory.

E

EA Extended Attribute. A small amount of data which can be retrieved through a name associated with a particular inode. Lustre uses EAa to store striping information (location of file data on OSTs). Examples of extended attributes are ACLs, striping information, and crypto keys.

Eviction The process of eliminating server state for a client that is not returning to the cluster after a timeout or if server failures have occurred.

Export The state held by a server for a client that is sufficient to transparently recover all in-flight operations when a single failure occurs.

Extent Lock A lock used by the OSC to protect an extent in a storage object for concurrent control of read/write, file size acquisition and truncation operations.

F

Failback The failover process in which the default active server regains control over the service.

Failout OST An OST which is not expected to recover if it fails to answer client requests. A failout OST can be administratively failed, thereby enabling clients to return errors when accessing data on the failed OST without making additional network requests.

Failover The process by which a standby computer server system takes over for an active computer server after a failure of the active node. Typically, the standby computer server gains exclusive access to a shared storage device between the two servers.

FID Lustre File Identifier. A collection of integers which uniquely identify a file or object. The FID structure contains a sequence, identity and version number.

Fileset A group of files that are defined through a directory that represents a file system's start point.

FLDB FID Location Database. This database maps a sequence of FIDs to a server which is managing the objects in the sequence.

Flight Group Group or I/O transfer operations initiated in the OSC, which is simultaneously going between two endpoints. Tuning the flight group size correctly leads to a full pipe.

G

Glimpse callback An RPC made by an OST or MDT to another system, usually a client, to indicate to tthat an extent lock it is holding should be surrendered if it is not in use. If the system is using the lock, then the system should report the object size in the reply to the glimpse callback. Glimpses are introduced to optimize the acquisition of file sizes.

Group Lock

Group upcall

I

Import The state held by a client to fully recover a transaction sequence after a server failure and restart.

Intent Lock A special locking operation introduced by Lustre into the Linux kernel. An intent lock combines a request for a lock, with the full information to perform the operation(s) for which the lock was requested. This offers the server the option of granting the lock or performing the operation and informing the client of the operation result without granting a lock. The use of intent locks enables metadata operations (even complicated ones), to be implemented with a single RPC from the client to the server.

IOV I/O vector. A buffer destined for transport across the network which contains a collection (a/k/a as a vector) of blocks with data.

K

Kerberos An authentication mechanism, optionally available in an upcoming Lustre version as a GSS backend.

L

LBUG A bug that Lustre writes into a log indicating a serious system failure.

LDLM Lustre Distributed Lock Manager.

lfs The Lustre File System configuration tool for end users to set/check file striping, etc. See lfs.

lfsck Lustre File System Check. A distributed version of a disk file system checker. Normally, lfsck does not need to be run, except when file systems are damaged through multiple disk failures and other means that cannot be recovered using file system journal recovery.

liblustre Lustre library. A user-mode Lustre client linked into a user program for Lustre fs access. liblustre clients cache no data, do not need to give back locks on time, and can recover safely from an eviction. They should not participate in recovery.

Llite Lustre lite. This term is in use inside the code and module names to indicate that code elements are related to the Lustre file system.

Llog Lustre log. A log of entries used internally by Lustre. An llog is suitable for rapid transactional appends of records and cheap cancellation of records through a bitmap.

Llog Catalog Lustre log catalog. An llog with records that each point at an llog. Catalogs were introduced to give llogs almost infinite size. llogs have an originator which writes records and a replicator which cancels record (usually through an RPC), when the records are not needed.

LMV Logical Metadata Volume. A driver to abstract in the Lustre client that it is working with a metadata cluster instead of a single metadata server.

LND Lustre Network Driver. A code module that enables LNET support over a particular transport, such as TCP and various kinds of InfiniBand, Elan or Myrinet.

LNET Lustre Networking. A message passing network protocol capable of running and routing through various physical layers. LNET forms the underpinning of LNETrpc.

Load-balancing MDSs A cluster of MDSs that perform load balancing of on system requests.

Lock Client A module that makes lock RPCs to a lock server and handles revocations from the server.

Lock Server A system that manages locks on certain objects. It also issues lock callback requests, calls while servicing or, for objects that are already locked, completes lock requests.

LOV Logical Object Volume. The object storage analog of a logical volume in a block device volume management system, such as LVM or EVMS. The LOV is primarily used to present a collection of OSTs as a single device to the MDT and client file system drivers.

LOV descriptor A set of configuration directives which describes which nodes are OSS systems in the Lustre cluster, providing names for their OSTs.

Lustre The name of the project chosen by Peter Braam in 1999 for an object-based storage architecture. Now the name is commonly associated with the Lustre file system.

Lustre client An operating instance with a mounted Lustre file system.

Lustre file A file in the Lustre file system. The implementation of a Lustre file is through an inode on a metadata server which contains references to a storage object on OSSs.

Lustre lite A preliminary version of Lustre developed for LLNL in 2002. With the release of Lustre 1.0 in late 2003, Lustre Lite became obsolete.

Lvfs A library that provides an interface between Lustre OSD and MDD drivers and file systems; this avoids introducing file system-specific abstractions into the OSD and MDD drivers.

M

Mballoc Multi-Block-Allocate. Lustre functionality that enables the ext3 file system to allocate multiple blocks with a single request to the block allocator. Normally, an ext3 file system only allocates only one block per request.

MDC MetaData Client - Lustre client component that sends metadata requests via RPC over LNET to the Metadata Target (MDT).

MDD MetaData Disk Device - Lustre server component that interfaces with the underlying Object Storage Device to manage the Lustre file system namespace (directories, file ownership, attributes).

MDS MetaData Server - Server node that is hosting the Metadata Target (MDT).

MDT Metadata Target. A metadata device made available through the Lustre meta-data network protocol.

Metadata Write-back Cache A cache of metadata updates (mkdir, create, setattr, other operations) which an application has performed, but have not yet been flushed to a storage device or server.

MGS Management Service. A software module that manages the startup configuration and changes to the configuration. Also, the server node on which this system runs.

Mountconf The Lustre configuration protocol (introduced in version 1.6) which formats disk file systems on servers with the mkfs.lustre program, and prepares them for automatic incorporation into a Lustre cluster.

N

NAL An older, obsolete term for LND.

NID Network Identifier. Encodes the type, network number and network address of a network interface on a node for use by Lustre.

NIO API A subset of the LNET RPC module that implements a library for sending large network requests, moving buffers with RDMA.

O

OBD Object Device. The base class of layering software constructs that provides Lustre functionality.

OBD API See Storage Object API.

OBD type Module that can implement the Lustre object or metadata APIs. Examples of OBD types include the LOV, OSC and OSD.

Obdfilter An older name for the OSD device driver.

Object device An instance of an object that exports the OBD API.

Object storage Refers to a storage-device API or protocol involving storage objects. The two most well known instances of object storage are the T10 iSCSI storage object protocol and the Lustre object storage protocol (a network implementation of the Lustre object API). The principal difference between the Lustre and T10 protocols is that Lustre includes locking and recovery control in the protocol and is not tied to a SCSI transport layer.

opencache A cache of open file handles. This is a performance enhancement for NFS.

Orphan objects Storage objects for which there is no Lustre file pointing at them. Orphan objects can arise from crashes and are automatically removed by an llog recovery. When a client deletes a file, the MDT gives back a cookie for each stripe. The client then sends the cookie and directs the OST to delete the stripe. Finally, the OST sends the cookie back to the MDT to cancel it.

Orphan handling A component of the metadata service which allows for recovery of open, unlinked files after a server crash. The implementation of this feature retains open, unlinked files as orphan objects until it is determined that no clients are using them.

OSC Object Storage Client. The client unit talking to an OST (via an OSS).

OSD Object Storage Device. A generic, industry term for storage devices with more extended interface than block-oriented devices, such as disks. Lustre uses this name to describe to a software module that implements an object storage API in the kernel. Lustre also uses this name to refer to an instance of an object storage device created by that driver. The OSD device is layered on a file system, with methods that mimic create, destroy and I/O operations on file inodes.

OSS Object Storage Server. A server OBD that provides access to local OSTs.

OST Object Storage Target. An OSD made accessible through a network protocol. Typically, an OST is associated with a unique OSD which, in turn is associated with a formatted disk file system on the server containing the storage objects.

P

Pdirops A locking protocol introduced in the VFS by CFS to allow for concurrent operations on a single directory inode.

pool OST pools allows the administrator to associate a name with an arbitrary subset of OSTs in a Lustre cluster. A group of OSTs can be combined into a named pool with unique access permissions and stripe characteristics.

Portal A concept used by LNET. LNET messages are sent to a portal on a NID. Portals can receive packets when a memory descriptor is attached to the portal. Portals are implemented as integers.

Examples of portals are the portals on which certain groups of object, metadata, configuration and locking requests and replies are received.

PTLRPC An RPC protocol layered on LNET. This protocol deals with stateful servers and has exactly-once semantics and built in support for recovery.

R

Recovery The process that re-establishes the connection state when a client that was previously connected to a server reconnects after the server restarts.

Reply The concept of re-executing a server request after the server lost information in its memory caches and shut down. The replay requests are retained by clients until the server(s) have confirmed that the data is persistent on disk. Only requests for which a client has received a reply are replayed.

Re-sent request A request that has seen no reply can be re-sent after a server reboot.

Revocation Callback An RPC made by an OST or MDT to another system, usually a client, to revoke a granted lock.

Rollback The concept that server state is in a crash lost because it was cached in memory and not yet persistent on disk.

Root squash A mechanism whereby the identity of a root user on a client system is mapped to a different identity on the server to avoid root users on clients gaining broad permissions on servers. Typically, for management purposes, at least one client system should not be subject to root squash.

routing LNET routing between different networks and LNDs.

RPC Remote Procedure Call. A network encoding of a request.

S

Storage Object API The API that manipulates storage objects. This API is richer than that of block devices and includes the create/delete of storage objects, read/write of buffers from and to certain offsets, set attributes and other storage object metadata.

Storage Objects A generic concept referring to data containers, similar/identical to file inodes.

Stride A contiguous, logical extent of a Lustre file written to a single OST.

Stride size The maximum size of a stride, typically 4 MB.

Stripe count The number of OSTs holding objects for a RAID0-striped Lustre file.

Striping metadata The extended attribute associated with a file that describes how its data is distributed over storage objects. See also default stripe pattern.

T

T10 object protocol An object storage protocol tied to the SCSI transport layer. Lustre does not use T10.

W

Wide striping Strategy of using many OSTs to store stripes of a single file. This obtains maximum bandwidth to a single file through parallel utilization of many OSTs.

Index

K

L

M

N

O

P

Q

R

S

T

U

V

W