Sun microsystems

Lustre™ 1.6 Operations Manual

Sun Microsystems, Inc.
www.sun.com

Part No. 820-3681-10

Lustre manual version: Lustre_1.6_man_v1.16

May 2009

g the Feedback[+] link at: http://docs.sun.com

Please
Recycle

Contents

Part IV Lustre for Users

Part V Reference

Preface

The *Lustre 1.6 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 1.6 Operations Manual	820-3681-10	A	November 2008	First Sun re-brand of Lustre manual.
Lustre 1.6 Operations Manual	820-3681-10	B	March 2008	Second Sun manual version.
Lustre 1.6 Operations Manual	820-3681-10	C	May 2008	Third Sun manual version.
Lustre 1.6 Operations Manual	820-3681-10	D	July 2008	Fourth Sun manual version.
Lustre 1.6 Operations Manual	820-3681-10	E	September 2008	Fifth Sun manual version.
Lustre 1.6 Operations Manual	820-3681-10	F	November 2008	Sixth Sun manual version.
Lustre 1.6 Operations Manual	820-3681-10	G	May 2009	Seventh Sun manual version.

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:

1.1 Introducing the Lustre File System

Lustre is a storage architecture for clusters. The central component is the Lustre file system, a shared file system for clusters. Currently, the Lustre file system is available for Linux and provides a POSIX-compliant UNIX file system interface. In 2008, a complementary Solaris version is planned.

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 (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 after the cluster is installed by adding servers dynamically.

Because Lustre is open source software, it has been adopted by numerous partners and integrated with their offerings. Both Red Hat and SUSE offer kernels with Lustre patches for easy deployment.

1.1.1 Lustre Key Features

The key features of Lustre include:

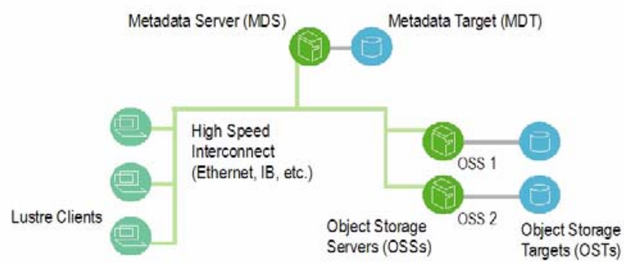
- **Scalability**: On Lustre, individual nodes, cluster size and disk storage are all scalable. For nodes, Lustre scales up and down well. For clusters, we currently support a production environment with 25,000 clients, and many clusters in the 10,000-20,000 client range are supported. Another installation supports 450 OSSs with up to 1,000 OSTs. For disk storage, several 1 PB Lustre file systems have been in use since 2006, with a 2 billion file maximum.
- **Performance**: On clusters, Lustre offers current performance of 100 GB/s in production deployments, 130 GB/s sustained in a test environment, and 13,000 creates/s sustained. On nodes, Lustre offers current single node performance of 2 GB/s client throughout (max) and 2.5 GB/s OSS throughput (max).
- **POSIX compliance**: The full POSIX test suite passes on Lustre clients. In a cluster, POSIX 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 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 ACLs are supported.
- **Open source**: Lustre is licensed under the GNU GPL

1.2 Lustre Components

A Lustre cluster consists of the following basic components:

- Metadata Server (MDS)
- Metadata Targets (MDT)
- Object Storage Servers (OSS)
- Object Storage Target (OST)
- Lustre clients

FIGURE 1-1 Lustre components in a basic cluster



1.2.1 MDS

The MDS is a server that makes metadata available to Lustre clients via MDTs. Each MDS manages the names and directories in the file system, and provides the network request handling for one or more local MDTs.[1]

1.2.2 MDT

The MDT stores metadata (such as filenames, directories, permissions and file layout) on an MDS. There is one MDT per file system.

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.

1.2.3 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[2] each.

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.

1.2.4 OST

The OST stores file data (chunks of user files) 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.

1. For historical reasons, the term “MDS” has traditionally referred to both the MDS and a single MDT. This manual version (and future versions) use the more specific meaning.

2. Lustre observes the IEC standard for base 2 and base 10 naming.

1.2.5 Lustre Clients

Lustre clients are computational, visualization or desktop nodes that mount the Lustre file system.[3]

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.

1.2.6 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. Lustre Network Drivers (LNDs) are available for a number of commodity and high-end networks, including TCP/IP, Quadrics Elan, Myrinet (MX and GM) and Cray.

1.2.7 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 to other Lustre components.

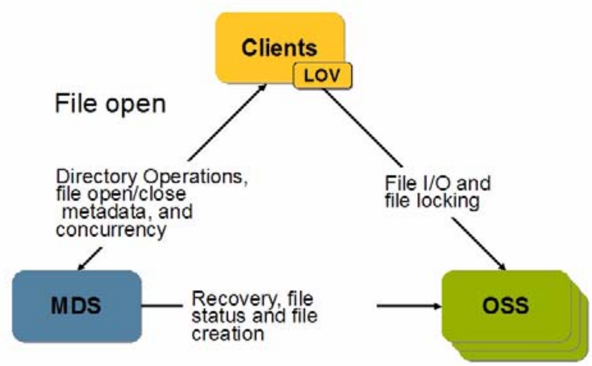
---

3. Lustre clients require Lustre software to mounta a Lustre file system.

1.3 Lustre Systems

Lustre components work together as coordinated systems to manage file and directory operations in the file system.

FIGURE 1-2 Lustre system interaction in a file system

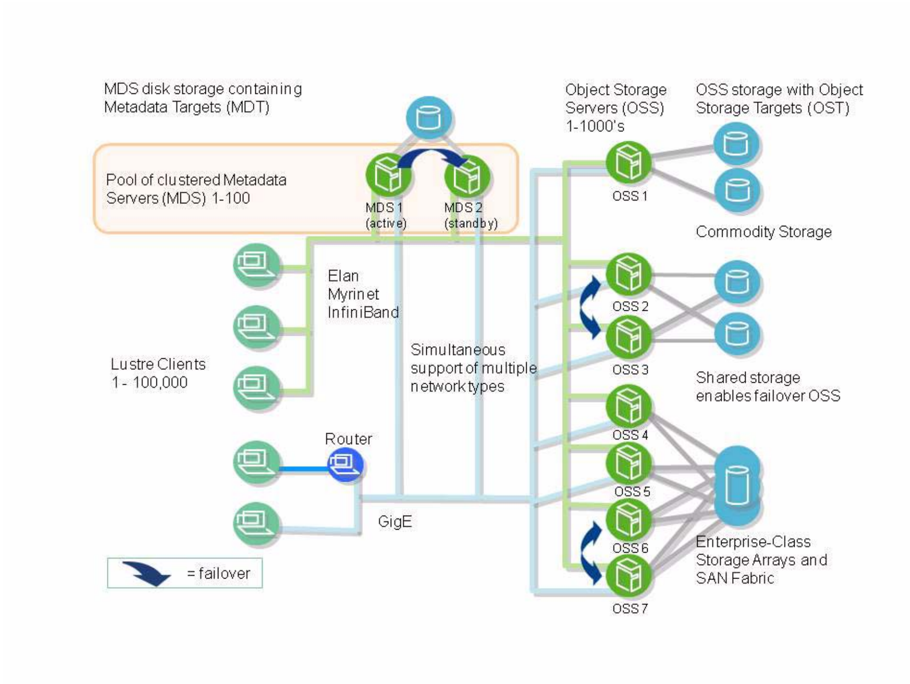


The characteristics of the Lustre system include:

	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.

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

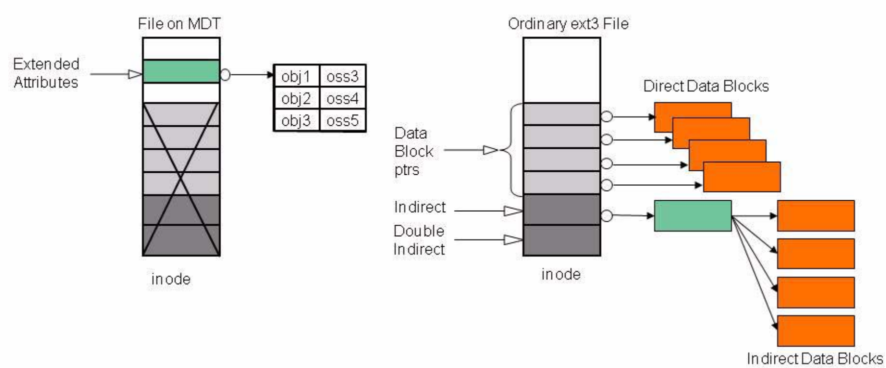
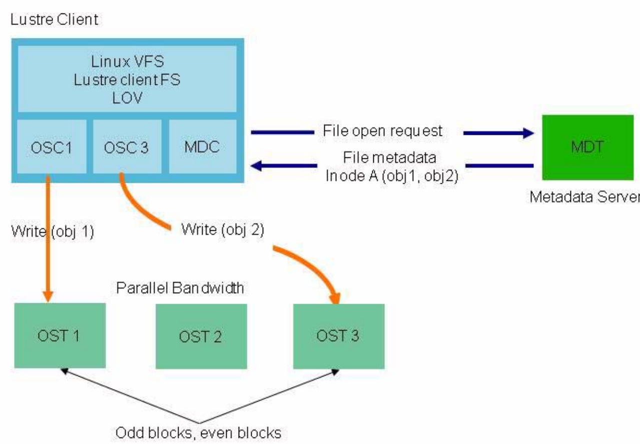


FIGURE 1-5 shows how a file open operation transfers the object pointers from the MDS to the client when a client opens the file, and how the client uses this information to perform I/O on the file, directly interacting with the OSS nodes where the objects are stored.

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



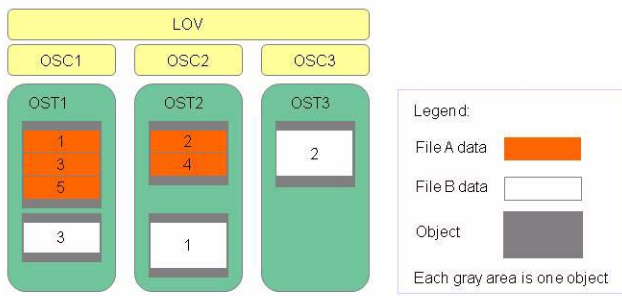
If only one object is associated with an MDS inode, that object contains all of the data in that Lustre file. When more than one object is associated with a file, data in the file is "striped" across the objects.

The 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 by a file. This leads to a maximum disk use of 1.48 PB by a file in Lustre. Note that the maximum file size is much larger (2^64 bytes), but the file cannot have more than 1.48 PB of allocated data; hence a file larger than 1.48 PB 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.

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
```

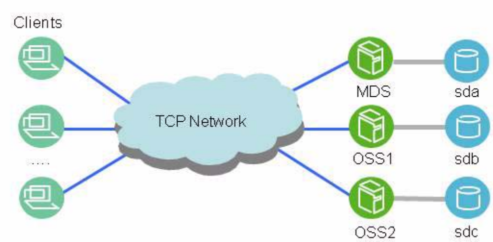
On OSS1:

```
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.

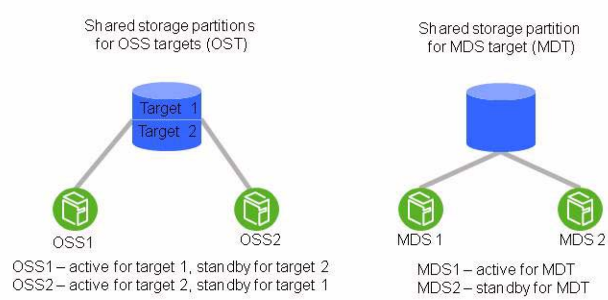
1.7 Lustre Failover and Rolling Upgrades

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.

1.8 Additional Lustre Features

Additional features of the Lustre file system are described below.

- **Interoperability**: Lustre runs on many CPU architectures (x86, IA-64, x86-64 (EM64 and AMD64, Power PC architectures [clients only], and mixed-endian clusters; clients and servers are interoperable between these platforms. Lustre strives to provide interoperability between adjacent software releases. Versions 1.4.x (x > 7) and version 1.6.x can interoperate with mixed clients and servers.[4]
- **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) for Lustre.[5]

Other current features of Lustre are described in detail in this manual. Future features are described in the Lustre roadmap.

4. Future Lustre releases may require "server first" or "all nodes at once" upgrade scenarios.
5. 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.

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 .

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 Modprobe.conf.

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.

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 (`'`). Otherwise, the command will not work.

2.4 Configuring LNET

This section describes how to configure LNET.

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 network 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, for example:

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

This specifies that the node should use a TCP interface and an Elan interface.

All LNET routers that bridge two networks are equivalent. Their configuration is not primary or secondary. All available routers balance their overall load. Router fault tolerance only works from Linux nodes. 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 for general use. It was created to work with specific hardware (Cray) and should never be used with other hardware.

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, see Modprobe.conf.

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

Caution – For options `ip2nets`, `routes` and `networks`, several best practices must be followed or configuration errors occur:

Best Practice 1: If you add a comment to any of the options mentioned above, 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 parmeter. 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.

Router fault tolerance only works from Linux nodes, that is, service nodes and application nodes if they are running Compute Node Linux (CNL). For this, LNET routing must correspond exactly with the Linux nodes' map of alive routers.[1]

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.

1. Catamount applications need an environmental variable set to configure LNET routing, which must correspond exactly to the Linux nodes' map of alive routers. The Catamount application must establish connections to all routers before the server replies (load-balanced over available routers), to be guaranteed to be routed back to them.

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 if the local node's networks are set up correctly.

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

Lustre Installation

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 400MB 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 sucessfully install and run Lustre, make sure the following installation prerequisites have been met:

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

3.1.1 Supported Operating System, Platform and Interconnect

Lustre supports the following operating systems, platforms[2] and interconnects. Make sure you are using a supported configuration.

Configuration Component	Supported Type
Operating system	Red Hat Enterprise Linux 4 and 5 SuSE Linux Enterprise Server 9 and 10 Linux 2.6, and a higher kernel than 2.6.15
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)

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

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).

3.1.2 Required Tools and Utilities

The Lustre software includes several tools needed for setup and monitoring; several third-party utilities are also required.

Note – Most of these tools and utilities are provided in the Lustre RPMs.

The Lustre utilites include:

- **lctl** - Low-level configuration utility that can be used to troubleshoot and debug Lustre.
- **lfs** - Used to read/set information about the Lustre file system's usage, such as striping, quota, OSTs, etc.
- **mkfs.lustre** - Formats Lustre target disks.
- **mount.lustre** - Lustre-specific helper for mount(8).
- **LNET self-test** - Helps determine that LNET and the network software and hardware are performing as expected.

Lustre requires several third-party tools be installed:

- **e2fsprogs**: Lustre requires very modern versions of e2fsprogs that understand extents. Use `e2fsprogs-1.38-<ver>` or later, available with the Lustre file downloads.

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 modern Perl should work with Lustre.
- **Build tool/Compiler** - If you plan to build Lustre from source code, then you need a GCC compiler; use GCC 3.0 or later. If you are installing Lustre from RPMs, you do not need a compiler.

3.1.3 High-Availability Software

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

Heartbeat supports a redundant system with access to the Shared (Common) Storage with dedicated connectivity; it can determine the system's general state. For more information, see Failover.

3.1.4 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 about these third-party tools, see the following 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/

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

4. ldiskfs is the Sun development version of ext4.

3.1.5 Environmental Requirements

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

Pdsh or SSH Access

Although not strictly required to run Lustre, we recommend that all cluster nodes have remote shell client access (preferably Pdsh[5], although SSH[6] is acceptable), to facilitate the use of Lustre configuration and monitoring scripts. For more information, see Pdsh.

Consistent Clocks

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 be a problem. It can also be 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. All machines in the cluster should synchronize their time from a local time server (or servers) at a suitable time interval.

For more information about NTP, see:

http://www.ntp.org/

Universal UID / GID

Maintain uniform file access permissions on all cluster nodes by using 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.

5. Parallel Distributed SHell

6. Secure SHell

3.1.6 Memory Requirements

This section describes the memory requirements of Lustre.

3.1.6.1 Determining the MDS's Memory

Use the following factors to determine the MDS's memory:

- 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

This suggests a minimum RAM size of 4 GB, but having more RAM is always prudent given the relatively low cost of this single component compared to the total system cost.

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.

3.1.6.2 OSS Memory Requirements

When planning the hardware for an OSS node, consider the memory usage of several components in the Lustre system. Although Lustre versions 1.4 and 1.6 do not cache file data in memory on the OSS node, there are a number of large memory consumers that need to be taken into account. Also consider that future Lustre versions will cache file data on the OSS node, so these calculations should only be taken as a minimum requirement.

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. In addition, 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. Also, 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. Finally, if you are using TCP or other network transport that uses system memory for send/receive buffers, this must also be taken into consideration.

Also, if the OSS nodes are to 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 Memory Usage for a 2 OST server (major consumers):

- 400MB journal size * 2 OST devices = 800MB
- 1.5MB per OST IO thread * 256 threads = 384MB
- e1000 RX descriptors, RxDescriptors=4096 for 9000 byte MTU = 128MB

This consumes over 1,300 MB just for the pre-allocated buffers, and does not include memory for the OS or file system metadata. For a non-failover configuration, 2 GB of RAM would be the minimum. For a failover configuration, 3 GB of RAM would be the minimum.

3.2 Installing Lustre from RPMs

Once all prerequisites have been met, you are ready to install Lustre.

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 (on the MDS, MGS and OSSs) must be patched; it is optional whether to patch the kernel on the Lustre clients. You can run the patched server kernel on the clients, but it is not necessary unless the clients will be used for multiple purposes, for example, to run as 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, exercise caution and back up ALL data.

---

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 RPMs/tarballs.**

   a. **Navigate to the Lustre download site and 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, using either the Sun Download Manager (SDM) or downloading the files individually.**

Tip – When considering where to install Lustre clients and servers, remember that for best performance in a production environment, dedicated clients are always best.

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.

Regarding servers, the MDS and MGS can be run together on the same machine.

If you are setting up a non-production Lustre environment, conducting testing, performing quick sanity tests, etc., it is okay to run Lustre clients and servers on the same node.

3. **Install the Lustre packages.**

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

   a. **For each Lustre package, determine if it needs to be installed on servers and/or clients.**

      TABLE 3-1 lists the Lustre packages. Use this table 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 packages, descriptions and installation guidance

Lustre Package	Description	Install on servers	Install on patchless clients	Install on patched clients
Lustre kernel RPMs				
kernel-lustre-smp-<ver>	Lustre-patched kernel package.	X		X*
kernel-lustre-bigsmp-<ver>	Lustre-patched kernel package for use on SuSE Server 9 and 10, i686 platforms.	X		X*
kernel-ib-<ver>	Lustre OFED package. Install if the network interconnect is InfiniBand (IB).	X	X	X*
Lustre module RPMs				
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 patchess 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 -i command to install the e2fsprogs package. For example:

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

If you want to add any optional packages to your Lustre file system, install them now.

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

3.3 Installing Lustre from Source Code

Installing Lustre from source involves several procedures - patching the core kernel, configuring it to work with Lustre, and creating Lustre and kernel RPMs from source code. The easier installation method is to install Lustre from packaged binaries (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, which provides an example 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. Patches can be applied from the stack (**quilt push**) or removed from the stack (**quilt pop**). You can 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**). You can 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 here:

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

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

3.3.1.2 Get the Lustre Source and Unpatched Kernel

The Lustre Group supports several Linux unpatched kernels for use with Lustre 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. The unpatched kernels are also available for download.

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

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

2. **Get the Lustre source code. Navigate to the Lustre** download site**, select the Lustre version you want and Source as the platform.**

   The files required to install Lustre from source code (unpatched kernels, Lustre source and e2fsprogs) are listed.

3. **Download the Lustre source code (lustre-<ver>.tar.gz).**

4. **Download the unpatched kernel you want to use.**

   If you do not know the kernel's filename, check the which_patch file.

   a. **In the Lustre source file, navigate to the which_patch file (lustre/kernel_patches/which_patch) and get the filename of the kernel you want to use.**

      The `which_patch` file lists the kernels supported in this release.

   b. **Download the selected kernel from the same location where you downloaded the Lustre source in Step 2.**

5. **To save time later, download the e2fsprogs tarball (e2fsprogs-<ver>tar.gz).**

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.**

   Lustre source and the unpatched kernel were previously downloaded in Get the Lustre Source and Unpatched Kernel.

   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:

```
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 opening a 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/` directoryof 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:

```
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.[7] For guidance on where to install specific packages, see TABLE 3-1. Also, Lustre packages should be installed in a specific 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 5 is unpacked, and use the rpm -i command to install it. For example:

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

   If you want to add any optional packages to your Lustre file 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.

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

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 Tools and Utilities.**

   ```
   $ 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

This chapter describes how to configure Lustre and includes the following sections:

- Configuring Lustre
- Basic Lustre Administration
- Operational Scenarios

4.1 Configuring Lustre

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. **Create a combined 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.

---

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

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

4. **Create one or more OSTs[2] for an OSS. For each OST, run this command on the OSS node:**

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

You can have as many OSTs per OSS as the hardware/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 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.

---

5. **Mount the OSTs. For each OST, run this command on the OSS node where the OST was created:**

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

6. **Mount the file system on the client. On the client node, run:**

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

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

   a. **Run the `df` command.**

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

   b. **Run the `dd` command.**

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

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

   c. **Run the `ls` command.**

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

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

---

2. When you create an 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.

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

If you are configuring Lustre for the first time or want to follow the steps in a simple test installation, use this configuration example, where:

Variable	Setting	Variable	Setting
network type	TCP/IP	**MGS node**	10.2.0.1@tcp0
block device	/dev/loop0	**OSS 1 node**	oss1
file system	temp	**OSS 2 node**	oss2
mount point	/mnt/mdt	**client node**	client1
mount point	/lustre	**OST 1**	ost1
		OST 2	ost2

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/loop0
```

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/loop0
      target name  temp-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/loop0
Writing CONFIGS/mountdata
```

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

```
[root@mds /]# mount -t lustre /dev/loop0 /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/loop0 has started
```

4. **Create the OSTs.**

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

   a. **Create ost1. On the oss1 node, run:**

```
[root@oss1 /]# mkfs.lustre --ost --fsname=temp --mgsnode=
10.2.0.1@tcp0 /dev/loop0
```

This 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/loop1
       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/loop1
Writing CONFIGS/mountdata
```

b. **Create ost2. On the oss2 node, run:**

```
[root@oss2 /]# mkfs.lustre --ost --fsname=temp --mgsnode=
10.2.0.1@tcp0 /dev/loop0
```

This 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/loop1
       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/loop1
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 the oss1 node, run:**

```
root@oss1 /] mount -t lustre /dev/loop0 /mnt/ost1
```

   This 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/loop0 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 the oss2 node, run:**

```
[root@oss2 /] mount -t lustre /dev/loop0 /mnt/ost2
```

   This command generates this output:

```
LDISKFS-fs: file extents enabled
LDISKFS-fs: mballoc enabled
Lustre: temp-OST0001: new disk, initializing
Lustre: Server temp-OST0001 on device /dev/loop0 has started
```

   Shortly afterwards, this output appears:

```
Lustre: temp-OST0001: received MDS connection from 10.2.0.1@tcp0
Lustre: MDS temp-MDT0000: temp-OST0001_UUID now active, resetting
orphans
```

6. **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 UNIX commands `df`, `dd` and `ls` on the client node.**

   a. **Run the `df` command:**

```
[root@client1 /] 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)

4.1.0.3 Lustre Configuration Utilities

Several configuration utilities are available to help you configure Lustre. For man pages and reference information, see:

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

The System Configuration Utilities (man8) chapter also includes information on other utilities, such as lustre_rmmod.sh, e2scan, l_getgroups, llobdstat, llstat, lst, plot-llstat, routerstat, and ll_recover_lost_found_objs, as well as utilities to manage large clusters, perform application profiling, and test and debug Lustre.

4.2 Basic Lustre Administration

Once you have the Lustre system up and running, use the basic administration procedures in this section to specify a file system name, start and stop servers, find nodes in a file system, remove an OST, etc. This section contains the following procedures:

- Specifying the File System Name
- 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
- Running the Writeconf Command
- Removing and Restoring OSTs
- Changing a Server NID
- Aborting Recovery
- Failover
- Unmounting a Server (without Failover)
- Unmounting a Server (with Failover)
- Changing the Address of a Failover Node

4.2.1 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 <block device name> <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>
```

4.2.2 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.

4.2.3 Unmounting a Server

Stopping a Lustre server is simple and only requires the `umount` command.

```
umount <mount point>
```

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

```
$ umount /mnt/test/ost0
```

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 `-f` ("force") flag is given, 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.

4.2.4 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`.

4.2.5 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
```

4.2.6 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
```

4.2.7 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" <mount point>
```

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>
```

4.2.8 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 the mkfs.lustre command creates a file system named lustre. To specify a different file system name[3], run:

```
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 /dev/sda, run:

```
mount -t lustre mgsnode:/foo /dev/sda
```

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.

---

3. Note that the file system name is limited to 8 characters.

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 /dev/sda and /dev/sdb.

```
mgsnode# mkfs.lustre --mgs /dev/sda
mdtfoonode# mkfs.lustre --fsname=foo --mdt --mgsnode=mgsnode@tcp0 /dev/sda
ossfoonode# mkfs.lustre --fsname=foo --ost --mgsnode=mgsnode@tcp0 /dev/sda
ossfoonode# mkfs.lustre --fsname=foo --ost --mgsnode=mgsnode@tcp0 /dev/sdb
mdtbarnode# mkfs.lustre --fsname=bar --mdt --mgsnode=mgsnode@tcp0 /dev/sda
ossbarnode# mkfs.lustre --fsname=bar --ost --mgsnode=mgsnode@tcp0 /dev/sda
ossbarnode# mkfs.lustre --fsname=bar --ost --mgsnode=mgsnode@tcp0 /dev/sdb
```

To mount a client on file system foo at mount point /dev/sda, run:

```
mount -t lustre mgsnode@tcp0:/foo /dev/sda
```

To mount a client on file system bar at mount point /dev/sdb, run:

```
mount -t lustre mgsnode@tcp0:/bar /dev/sdb
```

4.2.9 Running the Writeconf Command

If the system's configuration logs are in a state where the file system cannot be started or if you are changing a server NID, use the writeconf command to erase all of the file system's configuration logs (including all lctl conf_param settings).

After the writeconf command is run, the configuration logs are re-generated as servers restart, and the current server NIDs are used.

To run the writeconf command:

1. **Unmount all servers and clients.**
2. **On the MDT, run:**

   ```
   $ mdt> tunefs.lustre --writeconf <mount point>
   ```

3. **Remount all servers. You must mount the MDT *first*.**

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.

4.2.10 Removing and Restoring OSTs

OSTs can be removed from and restored to a Lustre file system.

4.2.10.1 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 so no new objects are allocated on the corresponding OST. 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
   ```

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. **Use** `lfs find` **to discover all files that have objects residing on the deactivated OST.**

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
```

4.2.10.2 Restoring an OST to 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. Run:**

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

4.2.11 Changing a Server NID

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.**

2. **Use the writeconf command to erase the configuration logs for the file system. On the MDT, run:**

```
$ mdt> tunefs.lustre --writeconf <mount point>
```

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

3. **If the MGS's NID was changed, communicate the new MGS location to each server. Run:**

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

4.2.12 Aborting Recovery

When starting a target, to abort the recovery process, 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 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 multiple 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
```

4.3.1 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/
```

4.4 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>
```

4.4.1 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.

4.4.2 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.

4.4.3 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.**

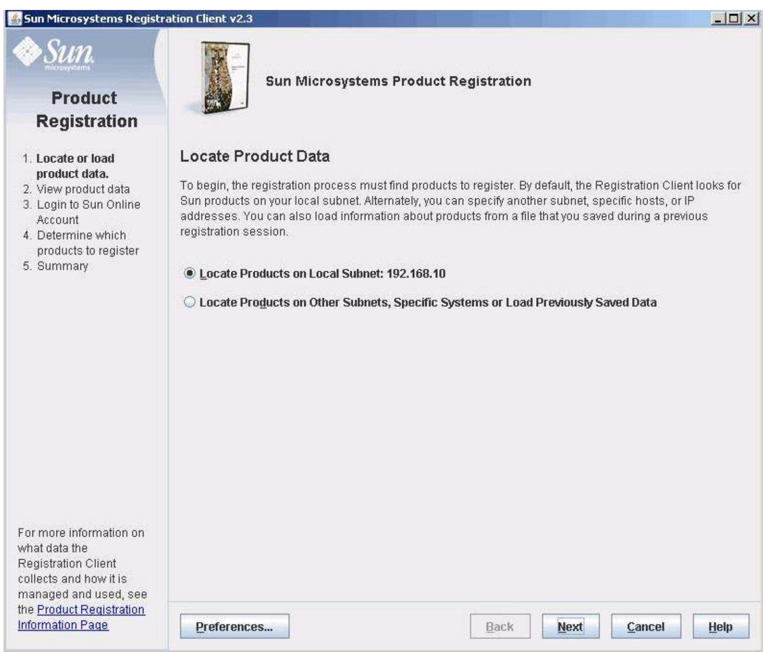
   Java VM is available at the Sun Java download site.

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.

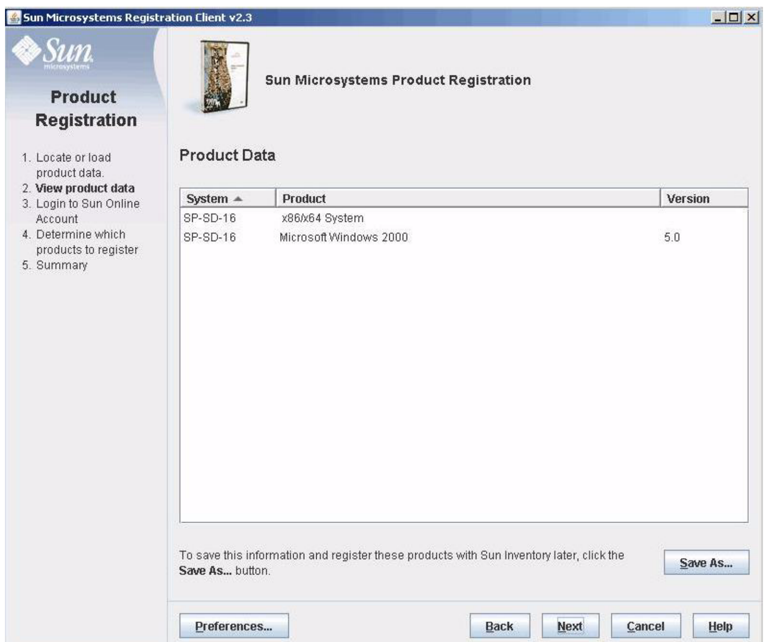
The registration process includes up to five steps. The first step is to discover the service tags created when you started Lustre.

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:

- Simple TCP Network

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 Lustre Installation) 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 Lustre Installation) 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. 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:

- Multi-homed Servers
- Elan to TCP Routing
- Load Balancing with InfiniBand
- Multi-Rail Configurations with LNET

7.1 Multi-homed 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

There is one OSS with two InfiniBand HCAs. Lustre clients have only one InfiniBand HCA using native Lustre drivers of o2ibind. Load balancing is done on both HCAs on the OSS with the help of LNET.

7.3.1 Modprobe.conf

Lustre users have options available on following networks.

- 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]
  ```

7.3.2 Start servers

To start the MGS and MDT server, run:

```
modprobe lnet
```

To start MGS and MDT, run:

```
$ 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
```

To start the OSS, run:

```
$ 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
```

7.3.3 Start clients

For the IB client, run:

```
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 bandwith 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 loss of a server or a network or software fault, the services provided remain unaffected. Generally, we measure availability by the percentage of time the system is required to be available.

Availability is accomplished by providing replicated hardware and/or software, so failure of the system will be covered by a paired system. The concept of “failover” is the method of switching an application and its resources to a standby server when the primary system fails or is unavailable. Failover should be automatic and, in most cases, completely application-transparent.

In Lustre, failover means that a client that tries to do I/O to a failed OST continues to try (forever) until it gets an answer. A userspace sees nothing strange, other than that I/O takes (potentially) a very long time to complete.

Lustre failover requires two nodes (a failover pair), which must be connected to a shared storage device. Lustre supports failover for both metadata and object storage servers. Failover is achieved most simply by powering off the node in failure (to be absolutely sure of no multi-mounts of the MDT) and mounting the MDT on the partner. When the primary comes back, it MUST NOT mount the MDT while secondary has it mounted. The secondary can then unmount the MDT and the master mount it.

The Lustre file system only supports failover at the server level. Lustre does not provide the tool set for system-level components that is needed for a complete failover solution (node failure detection, power control, and so on).[1]

Lustre failover is dependant on either a primary or backup OST to recover the file system. You need to set up an external HA mechanism. The recommended choice is the Heartbeat package, available at:

www.linux-ha.org

Heartbeat is responsible to detect failure of the primary server node and control the failover. The HA software controls Lustre using its built-in "file system" mechanism to unmount and mount file systems. Although Heartbeat is recommended, Lustre works with any HA software that supports resource (I/O) fencing.

The hardware setup requires a pair of servers with a shared connection to a physical storage (like SAN, NAS, hardware RAID, SCSI and FC). The method of sharing storage should be essentially transparent at the device level, that is, the same physical LUN should be visible from both nodes. To ensure high availability at the level of physical storage, we encourage the use of RAID arrays to protect against drive-level failures.

To have a fully-automated, highly-available Lustre system, you need power management software and HA software, which must provide the following -

- **Resource fencing** - Physical storage must be protected from simultaneous access by two nodes.
- **Resource control** - Starting and stopping the Lustre processes as a part of failover, maintaining the cluster state, and so on.
- **Health monitoring** - Verifying the availability of hardware and network resources, responding to health indications given by Lustre.

1. This functionality has been available for some time in third-party tools.

For proper resource fencing, the Heartbeat software must be able to completely power off the server or disconnect it from the shared storage device. It is imperative that no two active nodes access the same storage device, at the risk of severely corrupting data. When Heartbeat detects a server failure, it calls a process (STONITH) to power off the failed node; and then starts Lustre on the secondary node using its built-in "file system" resource manager.

Servers providing Lustre resources are configured in primary/secondary pairs for the purpose of failover. When a server `umount` command is issued, the disk device is set read-only. This allows the second node to start service using that same disk, after the command completes. This is known as a *soft* failover, in which case both the servers can be running and connected to the net. Powering off the node is known as a *hard* failover.

8.1.1 The Power Management Software

The Linux-HA package includes a set of power management tools, known as STONITH (Shoot The Other Node In The Head). STONITH has native support for many power control devices, and is extensible. It uses expect scripts to automate control. PowerMan, by the Lawrence Livermore National Laboratory (LLNL), is a tool for manipulating remote power control (RPC) devices from a central location. Several RPC varieties are supported natively by PowerMan.

The latest versions of PowerMan are available at:

http://sourceforge.net/projects/powerman

For more information on PowerMan, go to:

https://computing.llnl.gov/linux/powerman.html

8.1.2 Power Equipment

A multi-port, Ethernet addressable RPC is relatively inexpensive. For recommended products, refer to the list of supported hardware on the PowerMan site. Linux Network Iceboxes are also very good tools. They combine the remote power control and the remote serial console into a single unit.

8.1.3 Heartbeat

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. For more information, see:

http://linux-ha.org/HeartbeatProgram

To download Linux-HA, go to:

http://linux-ha.org/download

Lustre supports both Heartbeat V1 and Heartbeat V2. V1 has a simpler configuration and works very well. V2 adds monitoring and supports more complex cluster topologies. For additional information, we recommend that you refer to the Linux-HA website.

8.1.4 Connection Handling During Failover

A connection is alive when it is active and in operation. When a connection request is sent, a connection is not established until either a reply arrives or a connection disconnects or fails. If there is no traffic on a given connection, periodically check the connection to ensure its status.

If an active connection disconnects, it leads to at least one timeout request. New and old requests are in sleep until:

- The reply arrives (in case of re-activation of the connection and during the re-send request asynchronously).
- The application gets a signal (such as TERM or KILL).
- The server evicts the client, which gives an I/O error (EIO) for these requests or the connection becomes "failed."

A timeout is effectively infinite. Lustre waits as long as it needs to avoid giving the application an EIO.

Note – A client process waits indefinitely until the OST is back alive, unless either the process is killed (which should be possible after the Lustre recovery timeout is exceeded, 100s by default), or the OST is explicitly marked "inactive" on the clients:

```
lctl --device <failed OSC device on the client> deactivate
```

After the OSC is marked inactive, all I/O to this OST should immediately return with -EIO, and not hang.

Note – Under heavy load, clients may have to wait a long time for requests sent to the server to complete (100s of seconds in some cases). It is difficult for clients to distinguish between heavy server load (common) and server death (unlikely).

In the case where a server dies and fails over, the clients have to wait for their requests to time out, then they resend and wait again (in the common case the server is just overloaded), then they try to contact another server listed as a failover server for that node.

If a connection goes to the "failed" condition, which happens immediately in "failout" OST mode, new and old requests receive EIOs. In non-failout mode, a connection can only get into this state by using `lctl deactivate`, which is the only option for the client in the event of an OST failure.

Failout means that if an OST becomes unreachable (because it has failed, been taken off the network, unmounted, turned off, etc.), then I/O to get objects from that OST cause a Lustre client to get an EIO.

8.1.5 Roles of Nodes in a Failover

A failover pair of nodes can be configured in two ways – **active / active** and **active / passive**. An active node actively serves data while a passive node is idle, standing by to take over in the event of a failure. In the following example, using two OSTs (both of which are attached to the same shared disk device), the following failover configurations are possible:

- **active / passive** - This configuration has two nodes out of which only one is actively serving data all the time.

  In case of a failure, the other node takes over.If the active node fails, the OST in use by the active node will be taken over by the passive node, which now becomes active. This node serves most services that were on the failed node.

- **active / active** - This configuration has two nodes actively serving data all the time. In case of a failure, one node takes over for the other.

  To configure this for the shared disk, the shared disk must provide multiple partitions; each OST is the *primary* server for one partition and the *secondary* server for the other partition. The **active / passive** configuration doubles the hardware cost without improving performance, and is seldom used for OST servers.

8.2 OST Failover

The OST has two operating modes: **failover** and **failout**. The default mode is failover. In this mode, the clients reconnect after a failure, and the transactions, which were in progress, are completed. Data on the OST is written synchronously, and the client replays uncommitted transactions after the failure.

In the failout mode, when any communication error occurs, the client attempts to reconnect, but is unable to continue with the transactions that were in progress during the failure. Also, if the OST actually fails, data that has not been written to the disk (still cached on the client) is lost. Applications usually see an EIO for operations done on that OST until the connection is reestablished. However, the LOV layer on the client avoids using that OST. Hence, the operations such as file creates and fsstat still succeed. The failover mode is the current default, while the failout mode is seldom used.

8.3 MDS Failover

The MDS has only one failover mode: active / passive, as only one MDS may be active at a given time. The failover setup is two MDSs, each with access to the same MDT. Either MDS can mount the MDT, but not both at the same time.

8.4 Configuring MDS and OSTs for Failover

8.4.1 Configuring Lustre for Failover

To add a failover partner to a Lustre configuration, use the `--failnode` option. This may be done at creation time with with `mkfs.lustre` or at a later time with `tunefs.lustre`. For a failover example, see More Complicated Configurations. For an explanation of the `mkfs.lustre` and `tunefs.lustre` utilities, see mkfs.lustre and tunefs.lustre.

8.4.2 Starting/Stopping a Resource

You can start a resource with the `mount` command and stop it with the `umount` command. For details, see Unmounting a Server.

8.4.3 Active/Active Failover Configuration

With OST servers it is possible to have a load-balanced active/active configuration. Each node is the primary node for a group of OSTs, and the failover node for other groups. To expand the simple two-node example, we add ost2 which is primary on nodeB, and is on the LUNs `nodeB:/dev/sdc1` and `nodeA:/dev/sdd1`. This demonstrates that the /dev/ identity can differ between nodes, but both devices must map to the same physical LUN. In this type of failover configuration, you can mount two OSTs on two different nodes, and format them from either node. With failover, two OSSs provide the same service to the Lustre network in parallel. In case of disaster or a failure in one of the nodes, the other OSS can provide uninterrupted file system services.

For an active/active configuration, mount one OST on one node and another OST on the other node. You can format them from either node.

8.4.4 Hardware Requirements for Failover

This section describes hardware requirements that must be met to configure Lustre for failover.

8.4.4.1 Hardware Preconditions

- The setup must consist of a failover pair where each node of the pair has access to shared storage. If possible, the storage paths should be identical (`nodeA:/dev/sda == nodeB:/dev/sda`).

Note – A failover pair is a combination of two or more separate nodes. Each node has access to the same shared disk.

- Shared storage can be arranged in an active/passive (MDS, OSS) or active/active (OSS only) configuration. Each shared resource has a primary (default) node. Heartbeat assumes that the non-primary node is secondary for that resource.
- The two nodes must have one or more communication paths for Heartbeat traffic. A communication path can be:
  - Dedicated Ethernet
  - Serial live (serial crossover cable)

  Failure of all Heartbeat communication is not good. This condition is called “split-brain”. Heartbeat software resolves this situation by powering down one node.
- The two nodes must have a method to control one another's state; RPC hardware is the best choice. There must be a script to start and stop a given node from the other node. STONITH provides soft power control methods (SSH, meatware), but these cannot be used in a production situation.
- Heartbeat provides a remote ping service that is used to monitor the health of the external network. If you wish to use the ipfail service, then you must have a very reliable external address to use as the ping target. Typically, this is a firewall route or another very reliable network endpoint external to the cluster.

In Lustre, a disk failure is an unrecoverable error. For this reason, you must have reliable back-end storage with RAID.

Note – If a disk fails, requiring you to change the disk or resync the RAID, you can deactivate the affected OST, using `lctl` on the clients and MDT. This allows access functions to complete without errors (files on the affected OST will be of 0-length, however, you can save rest of your files).

8.5 Setting Up Failover with Heartbeat V1

This section describes how to set up failover with Heartbeat V1.

8.5.1 Installing the Software

1. **Install Lustre (see Installing Lustre from RPMs).**
2. **Install the RPMs that are required to configure Heartbeat.**

   The following packages are needed for Heartbeat V1. We used the 1.2.3-1 version. RedHat supplies v1.2.3-2. Heartbeat is available as an RPM or source.

   These are the Heartbeat packages, in order:

   - heartbeat-stonith -> `heartbeat-stonith-1.2.3-1.i586.rpm`
   - heartbeat-pils -> `heartbeat-pils-1.2.3-1.i586.rpm`
   - heartbeat itself -> `heartbeat-1.2.3-1.i586.rpm`

   You can find the above RPMs at:

   http://linux-ha.org/download/index.html#1.2.3
3. **Satisfy the installation prerequisites.**

   Heartbeat 1.2.3 installation requires following:

   - python
   - openssl
   - libnet-> `libnet-1.1.2.1-19.i586.rpm`
   - libpopt -> `popt-1.7-274.i586.rpm`
   - librpm -> `rpm-4.1.1-222.i586.rpm`
   - glib -> `glib-2.6.1-2.i586.rpm`
   - glib-devel -> `glib-devel-2.6.1-2.i586.rpm`

8.5.1.1 Configuring Heartbeat

This section describes basic configuration of Heartbeat with and without STONITH.

Note – LNET does not support virtual IP addresses. The IP address specified in the haresources file should be a 'dummy' address (valid, but unused). With later releases of Heartbeat, you may avoid the use of virtual IPs, but it is required in earlier releases.

Basic Configuration - Without STONITH

The http://linux-ha.org website has several guides covering basic setup and initial testing of Heartbeat; We suggest that you read them.

1. **Configure and test the Heartbeat setup before adding STONITH.**

   Let us assume there are two nodes, nodeA and nodeB. nodeA owns ost1 and nodeB owns ost2. Both the nodes are with dedicated Ethernet – eth0 having serial crossover link – /dev/ttySO. Consider that both nodes are pinging to a remote host – 192.168.0.3 for health.

2. **Create /etc/ha.d/ha.cf**

   - This file must be identical on both the nodes.
   - Follow the specific order of the directives.
   - Sample `ha.cf` file

   # Suggested fields - logging

   ```
   debugfile /var/log/ha-debug
   logfile /var/log/ha-log
   logfacility local0
   ```

   # Required fields - Timing

   ```
   keepalive 2
   deadtime 30
   initdead 120
   ```

   # If using serial Heartbeat

   ```
   baud 19200
   serial /dev/ttyS0
   ```

   # For Ethernet broadcast

   ```
   udpport 694
   bcast eth0
   ```

   # Use manual failback

   ```
   auto_failback off
   ```

 # Cluster members - name must match `hostname`

```
node oss161.clusterfs.com oss162. clusterfs.com
```

 # remote health ping

```
ping 192.168.16.1
respawn hacluster /usr/lib/heartbeat/ipfail
```

3. **Create /etc/ha.d/haresources**
   - This file must be identical on both the nodes.
   - It specifies a virtual IP address and a service.
   - Sample haresources

```
oss161.clusterfs.com 192.168.16.35 \
Filesystem::/dev/sda::/ost1::lustre

oss162.clusterfs.com 192.168.16.36 \
Filesystem::/dev/sda::/ost1::lustre
```

4. **Create /etc/ha.d/authkeys**
   - Copy the example from `/usr/share/doc/heartbeat-`*`<version>`*.
   - chmod the file '0600' – Heartbeat does not start if the permissions on this file are incorrect.
   - Sample authkeys files

```
auth 1
1 sha1 PutYourSuperSecretKeyHere
```

   a. **Start Heartbeat.**

```
[root@oss161 ha.d]# service heartbeat start
Starting High-Availability services:
[ OK ]
```

b. **Monitor the syslog on both nodes. After the initial deadtime interval, you should see the nodes discovering each other's state, and then they start the Lustre resources they own. You should see the startup command in the log:**

```
Aug 9 09:50:44 oss161 crmd: [4733]: info: update_dc: Set DC to
<null> (<null>)
Aug 9 09:50:44 oss161 crmd: [4733]: info: do_election_count_vote:
Election check: vote from oss162.clusterfs.com
Aug 9 09:50:44 oss161 crmd: [4733]: info: update_dc: Set DC to
<null> (<null>)
Aug 9 09:50:44 oss161 crmd: [4733]: info: do_election_check:
Still waiting on 2 non-votes (2 total)
Aug 9 09:50:44 oss161 crmd: [4733]: info: do_election_count_vote:
Updated voted hash for oss161.clusterfs.com to vote
Aug 9 09:50:44 oss161 crmd: [4733]: info: do_election_count_vote:
Election ignore: our vote (oss161.clusterfs.com)
Aug 9 09:50:44 oss161 crmd: [4733]: info: do_election_check:
Still waiting on 1 non-votes (2 total)
Aug 9 09:50:44 oss161 crmd: [4733]: info: do_state_transition:
State transition S_ELECTION -> S_PENDING [ input=I_PENDING cause=
C_FSA_INTERNAL origin=do_election_count_vote ]
Aug 9 09:50:44 oss161 crmd: [4733]: info: update_dc: Set DC to
<null> (<null>)
Aug 9 09:50:44 oss161 crmd: [4733]: info: do_dc_release: DC role
released
Aug 9 09:50:45 oss161 crmd: [4733]: info: do_election_count_vote:
Election check: vote from oss162.clusterfs.com
Aug 9 09:50:45 oss161 crmd: [4733]: info: update_dc: Set DC to
<null> (<null>)
Aug 9 09:50:46 oss161 crmd: [4733]: info: update_dc: Set DC to
oss162.clusterfs.com (1.0.9)
Aug 9 09:50:47 oss161 crmd: [4733]: info: update_dc: Set DC to
oss161.clusterfs.com (1.0.9)
Aug 9 09:50:47 oss161 cib: [4729]: info: cib_replace_notify:
Local-only Replace: 0.0.1 from <null>
Aug 9 09:50:47 oss161 crmd: [4733]: info: do_state_transition:
State transition S_PENDING -> S_NOT_DC [ input=I_NOT_DC cause=
C_HA_MESSAGE origin=do_cl_join_finalize_respond ]
Aug 9 09:50:47 oss161 crmd: [4733]: info: populate_cib_nodes:
Requesting the list of configured nodes
Aug 9 09:50:48 oss161 crmd: [4733]: notice: populate_cib_nodes:
Node: oss162.clusterfs.com (uuid:
00e8c292-2a28-4492-bcfc-fb2625ab1c61)
Sep 7 10:42:40 d1_q_0 heartbeat: info: Running \
/etc/ha.d/resource.d/ost1 start
```

In this example, ost1 is the shared resource. Common things to watch out for:

- If you configure two nodes as primary for one resource, then you will see both nodes attempt to start it. This is very bad. Shut down immediately and correct your HA resources files.
- If the commutation between nodes is not correct, both nodes may also attempt to mount the same resource, or will attempt to STONITH each other. There should be many error messages in syslog indicating a communication fault.
- When in doubt, you can set a Heartbeat debug level in ha.cf—levels above 5 produce huge volumes of data.

c. **Try some manual failover/ failback. Heartbeat provides two tools for this purpose (by default they are installed in /usr/lib/heartbeat) –**

- hb_standby [local|foreign] - Causes a node to yield resources to another node—if a resource is running on its primary node it is local, otherwise it is foreign.
- hb_takeover [local|foreign] - Causes a node to grab resources from another node.

Basic Configuration - With STONITH

STONITH automates the process of power control with the expect package. Expect scripts are very dependent on the exact set of commands provided by each hardware vendor, and as a result any change made in the power control hardware/firmware requires tweaking STONITH.

Much must be deduced by running the STONITH package by hand. STONITH has some supplied packages, but can also run with an external script. There are two STONITH modes:

- Single STONITH command for all nodes found in ha.cf:

```
-------/etc/ha.d/ha.cf-------------------
stonith <type> <config file>
```

- STONITH command per-node:

```
-------/etc/ha.d/ha.cf--------------------
stonith_host <hostfrom> <stonith_type> <params...>
```

You can use an external script to kill each node:

```
stonith_host nodeA external foo /etc/ha.d/reset-nodeB
stonith_host nodeB external foo /etc/ha.d/reset-nodeA
```

Here, foo is a placeholder for an unused parameter.

To get the proper syntax, run:

```
$ stonith -L
```

The above command lists supported models.

To list required parameters and specify the config file name, run:

```
$ stonith -l -t <model>
```

To attempt a test, run:

```
$ stonith -l -t <model> <fake host name>
```

This command also gives data on what is required. To test, use a real hostname. The external STONITH scripts should take the parameters {`start|stop|status`} and return 0 or 1.

STONITH _only happens when the cluster cannot do things in an orderly manner. If two cluster nodes can communicate, they usually shut down properly. This means many tests do not produce a STONITH, for example:

- Calling init 0 or **shutdown** or **reboot** on a node, orderly halt, no STONITH
- Stopping the heartbeat service on a node, again, orderly halt, no STONITH

You have to do something drastic (for example, killall -9 heartbeat) like pulling cables, or so on before you trigger STONITH.

Also, the alert script does a software failover, which halts Lustre but does not halt or STONITH the system. To use STONITH, edit the `fail_lustre.alert` script and add your preferred shutdown command after the line:

```
`/usr/lib/heartbeat/hb_standby local &`;
```

A simple method to halt the system is the sysrq method. Run:

```
$ !/bin/bash
```

This script forces a boot. Run:

```
$ 'echo s' = sync
$ 'echo u' = remount read-only
$ 'echo b' = reboot
$

SYST="/proc/sysrq-trigger"
if [ ! -f $SYST ]; then
          echo "$SYST not found!"
          exit 1
fi

$ sync, unmount, sync, reboot
echo s > $SYST
echo u > $SYST
echo s > $SYST
echo b > $SYST

exit 0
```

8.6 Using MMP

The multiple mount protection (MMP) feature protects the file system from being mounted more than one time simultaneously. If the file system is mounted, MMP also protects changes by e2fsprogs to the file system. This feature is very important in a shared storage environment (for example, when an OST and a failover OST share a partition).

The backing file system for Lustre, `ldiskfs`, supports the MMP mechanism. A block in the file system is updated by a `kmmpd` daemon at one second intervals, and a monotonically increasing sequence number is written in this block. If the file system is cleanly unmounted, then a special "clean" sequence is written in this block. When mounting a file system, ldiskfs checks if the MMP block has a clean sequence or not.

Even if the MMP block holds a clean sequence, ldiskfs waits for some interval to guard against the following situations:

- Under heavy I/O, it may take longer for the MMP block to be updated
- If another node is also trying to mount the same file system, there may be a 'race'

With MMP enabled, mounting a clean file system takes at least 10 seconds. If the file system was not cleanly unmounted, then mounting the file system may require additional time.

Note – The MMP feature is only supported on Linux kernel versions >= 2.6.9.

Note – The MMP feature is automatically enabled by mkfs.lustre for new file systems at format time if failover is being used and the kernel and e2fsprogs support it. Otherwise, the Lustre administrator has to manually enable this feature when the file system is unmounted. If failover is being used, the MMP feature is automatically enabled by `mkfs.lustre`.

- To determine if MMP is enabled: `dumpe2fs -h <device>|grep features`

Example: `dumpe2fs -h /dev/{mdtdev} | grep 'Inode count'`

- To manually disable MMP: `tune2fs -O ^mmp <device>`
- To manually enable MMP: `tune2fs -O mmp <device>`

If ldiskfs detects that a file system is being mounted multiple times, it reports the time when the MMP block was last updated, the node name and the device name.

8.7 Setting Up Failover with Heartbeat V2

This section describes how to set up failover with Heartbeat V2.

8.7.1 Installing the Software

1. **Install Lustre (see Installing Lustre from RPMs).**

2. **Install RPMs required for configuring Heartbeat.**

   The following packages are needed for Heartbeat (v2). We used the 2.0.4 version of Heartbeat.

   Heartbeat packages, in order:

   - heartbeat-stonith -> heartbeat-stonith-2.0.4-1.i586.rpm
   - heartbeat-pils -> heartbeat-pils-2.0.4-1.i586.rpm
   - heartbeat itself -> heartbeat-2.0.4-1.i586.rpm

   You can find all the RPMs at the following location:

   http://linux-ha.org/download/index.html#2.0.4

3. **Satisfy the installation prerequisites.**

   To install Heartbeat 2.0.4-1, you require:

   - Python
   - openssl
   - libnet-> `libnet-1.1.2.1-19.i586.rpm`
   - libpopt -> `popt-1.7-274.i586.rpm`
   - librpm -> `rpm-4.1.1-222.i586.rpm`
   - libtld- > `libtool-ltdl-1.5.16.multilib2-3.i386.rpm`
   - lingnutls -> `gnutls-1.2.10-1.i386.rpm`
   - Libzo -> `lzo2-2.02-1.1.fc3.rf.i386.rpm`
   - glib -> `glib-2.6.1-2.i586.rpm`
   - glib-devel -> `glib-devel-2.6.1-2.i586.rpm`

8.7.2 Configuring the Hardware

Heartbeat v2 runs well with an un-altered v1 configuration. This makes upgrading simple. You can test the basic function and quickly roll back if issues appear. Heartbeat v2 does not require a virtual IP address to be associated with a resource. This is good since we do not use virtual IPs.

Heartbeat v2 supports multi-node clusters (of more than two nodes), though it has not been tested for a multi-node cluster. This section describes only the two-node case. The multi-node setup adds a score value to the resource configuration. This value is used to decide the proper node for a resource when failover occurs.

Heartbeat v2 adds a resource manager (crm). The resource configuration is maintained as an XML file. This file is re-written by the cluster frequently. Any alterations to the configuration should be made with the HA tools or when the cluster is stopped.

8.7.2.1 Hardware Preconditions

- The basic cluster assumptions are the same as those for Heartbeat v1. For the sake of clarity, here are the preconditions:
- The setup must consist of a failover pair where each node of the pair has access to shared storage. If possible, the storage paths should be identical (`d1_q_0:/dev/sda == d2_q_0:/dev/sda`).
- Shared storage can be arranged in an active/passive (MDS,OSS) or active/active (OSS only) configuration. Each shared resource will have a primary (default) node. The secondary node is assumed.
- The two nodes must have one or more communication paths for heartbeat traffic. A communication path can be:
  - Dedicated Ethernet
  - Serial live (serial crossover cable)

  Failure of all heartbeat communication is not good. This condition is called "split-brain" and the heartbeat software will resolve this situation by powering down one node.
- The two nodes must have a method to control each other's state. The Remote Power Control hardware is the best. There must be a script to start and stop a given node from the other node. STONITH provides soft power control methods (ssh, meatware) but these cannot be used in a production situation.
- Heartbeat provides a remote ping service that is used to monitor the health of the external network. If you wish to use the ipfail service, you must have a very reliable external address to use as the ping target.

8.7.2.2 Configuring Lustre

Configuring Lustre for Heartbeat V2 is identical to the V1 case.

8.7.2.3 Configuring Heartbeat

For details on all configuration options, refer to the Linux HA website:

http://linux-ha.org/ha.cf

As mentioned earlier, you can run Heartbeat V2 using the V1 configuration. To convert from the V1 configuration to V2, use the `haresources2cib.py` script (typically found in `/usr/lib/heartbeat`).

If you are starting with V2, we recommend that you create a V1-style configuration and converting it, as the V1 style is human-readable. The heartbeat XML configuration is located at `/var/lib/heartbeat/cib.xml` and the new resource manager is enabled with the crm yes directive in `/etc/ha.d/ha.cf`. For additional information on CiB, refer to:

http://linux-ha.org/ClusterInformationBase/UserGuide

Heartbeat log daemon

Heartbeat V2 adds a logging daemon, which manages logging on behalf of cluster clients. The UNIX syslog API makes calls that can block, Heartbeat requires log writes to complete as a sign of health. This daemon prevents a busy syslog from triggering a false failover. The logging configuration has been moved to `/etc/logd.cf`, while the directives are essentially unchanged.

Basic configuration (No STONITH or monitor)

Assuming two nodes, d1_q_0 and d21_q_0:

- d1_q_0 owns ost-alpha
- d2_q_0 owns ost-beta
- dedicated Ethernet - eth0
- serial crossover link - /dev/ttySO
- remote host for health ping - 192.168.0.3

Use this procedure:

1. **Create the basic ha.cf and haresources files.**

   haresources no longer requires the dummy virtual IP address.

   This is an example of /etc/ha.d/haresouces

   ```
   oss161.clusterfs.com 192.168.16.35 \ Filesystem::/dev/sda::/ost1::lustre
   oss162.clusterfs.com 192.168.16.36 \ Filesystem::/dev/sda::/ost1::lustre
   ```

   Once you have these files created, you can run the conversion tool:

   ```
   $ /usr/lib/heartbeat/haresources2cib.py -c basic.ha.cf \
   basic.haresources > basic.cib.xml
   ```

2. **Examine the cib.xml file**

   The first section in the XML file is <*attributes*>. The default values should be fine for most installations.

   The actual resources are defined in the <primitive> section. The default behavior of Heartbeat is an automatic failback of resources when a server is restored. To avoid this, you must add a parameter to the <*primitive*> definition. You may also like to reduce the timeouts. In addition, the current version of the script does not correctly name the parameters.

   ```
   <cib generated="true" admin_epoch="0" epoch="0" num_updates="0" \
   have_quorum="true" ignore_dtd="false" num_peers="2"
   ccm_transition="1" cib-last- \ written="Thu Aug 9 09:50:12 2007">
               <configuration>
                   <crm_config/>
                       <nodes>
   <node id="00e8c292-2a28-4492-bcfs-fb2625ab1c61" \
   uname="oss162.spsoftware.com" type="normal" />
   <node id="e370be9a-24f4-46a5-99ac-41a88c5fa344" \
   uname="oss161.spsoftware.com" type="normal"/>
                       </nodes>
                       <resources/>
                       <constraints/>
               </configuration>
   </cib>
   ```

   a. **Copy the modified resource file to /var/lib/heartbeat/crm/cib.xml**

   b. **Start the Heartbeat software.**

   c. **After startup, Heartbeat re-writes the cib.xml, adding a <node> section and status information. Do not alter those fields.**

Basic Configuration – Adding STONITH

As per Basic configuration (No STONITH or monitor), the best way to do this is to add the STONITH options to ha.cf and run the conversion script. For more information, see:

http://linux-ha.org/ExternalStonithPlugins

8.7.3 Operation

In normal operation, Lustre should be controlled by the Heartbeat software. Start Heartbeat at the boot time. It starts Lustre after the initial dead time.

8.7.3.1 Initial startup

1. **Stop the Heartbeat software (if running).**

   If this is a new Lustre file system:

   ```
   $ mkfs.lustre --fsname=spfs --ost --failnode=oss162 \
   --mgsnode=mds16@tcp0 /dev/sdb (one)
   ```

2. **mount -t lustre /dev/sdb /mnt/spfs/ost/**
3. **/etc/init.d/heartbeat start on one node.**
4. **tail -f /var/log/ha-log to see progress.**
5. **After initdead, this node should start all Lustre objects.**
6. **/etc/init.d/heartbeat start on second node.**
7. **After heartbeat is up on both the nodes, failback the resources to the second node. On the second node, run:**

   ```
   $ /usr/lib/heartbeart/hb_takeover local
   ```

   You should see the resources stop on the first node, and start up on the second node

8.7.3.2 Testing

1. **Pull power from one node.**
2. **Pull networking from one node.**
3. **After Mon is setup, pull the connection between the OST and the backend storage.**

8.7.3.3 Failback

Normally, do the failback manually after determining that the failed node is now good. Lustre clients can work during a failback, but they are momentarily blocked.

Note – When formatting the MGS, the `--failnode` option is not available. This is because MGSs do not need to be told about a failover MGS; they do not communicate with other MGSs at any time. However, OSSs, MDSs and Lustre clients need to know about failover MGSs. MDSs and OSSs are told about failover MGSs with the `--mgsnode` parameter and/or using multi-NID mgsspec specifications. At mount time, clients are told about all MGSs with a multi-NID mgsspec specification. For more details on the multi-NID mgsspec specification and how to tell clients about failover MGSs, see the mount.lustre man page.

8.8 Considerations with Failover Software and Solutions

The failover mechanisms used by Lustre and tools such as Hearbeat are soft failover mechanisms. They check system and/or application health at a regular interval, typically measured in seconds. This, combined with the data protection mechanisms of Lustre, is usually sufficient for most user applications.

However, these *soft* mechanisms are not perfect. The Heartbeat poll interval is typically 30 seconds. To avoid a false failover, Heartbeat waits for a *deadtime* interval before triggering a failover. In normal case, a user I/O request should block and recover after the failover completes. But this may not always be the case, given the delay imposed by Heartbeat.

Likewise, the Lustre health_check mechanism does not provide perfect protection against any or all failures. It is a sample taken at a time interval, not something that brackets each and every I/O request.[2] There are a few places where health_check could generate a bad status:

- On a device basis if there are requests that have not been processed in a very long time (more than the maximum allowed timeout), a CERROR is printed:

  ```
  {service}: unhealthy - request has been waiting Ns
  ```

  Ns is the number of seconds. The CERROR displays a true value for Ns, for example "... request has been waiting 100s"

- If the backing file system has gone read-only due to file system errors
- On a per-device basis if any of the above failed, it is reported in the `/proc/fs/lustre/health_check` file:

  ```
  device {device} reported unhealthy
  ```

- If ANY device or service on the node is unhealthy, it also prints:

  ```
  NOT HEALTHY
  ```

- If ALL devices and services on the node are healthy, it prints:

  ```
  healthy
  ```

There will be cases where a user job will die prior to the HA software triggering a failover. You can certainly shorten timeouts, add monitoring, and take other steps to decrease this probability. But there is a serious trade-off – shortening timeouts increases the probability of false-triggering a busy system. Increasing monitoring takes the system resources, and can likewise cause a false trigger.

Unfortunately, *hard* failover solutions capable of catching failures in the sub-second range generally require special hardware. As a result, they are quite expensive.

Tip – Failover of the Lustre client is dependent on the `obd_timeout` parameter. The Lustre client does not attempt failover until the request times out. Then, the client tries resending the request to the original server (if again, an `obd_timeout` occurs). After that, the Lustre client refers to the import list for that target and tries to connect (in a round-robin manner) until one of the nodes replies. The timeouts for the connection are much lower (obd_timeout / 20, 5).

---

2. This is true for every HA monitor, not just the Lustre health_check.

CHAPTER 9

Configuring Quotas

This chapter describes how to configure quotas and includes the following section:

- Working with Quotas

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 a quota is enabled, it is enabled for all clients of the file system and turned on automatically at mount.

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. To enable quotas:

1. **If you have re-complied your Linux kernel, be sure that CONFIG_QUOTA and CONFIG_QUOTACTL are enabled (quota is enabled in all the 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` **ommand.**

```
$ 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 on the file system, it is automatically enabled for all file system clients.

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
```

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 introduces a new quota format (v2), for administrative quota files, with continued support for the old quota format (v1).[1] The `mdt.quota_type` parameter also handles ‘1’ and ‘2’ options to specify the version of Lustre quota that will be used. For example:

```
--param mdt.quota_type=ug1
--param mdt.quota_type=u2
```

In a future Lustre release, the v2 format will be added to operational quotas, with continued support for the v1 format. When v2 support is added, then the `ost.quota_type` parameter will handle the ‘1’ and ‘2’ options.

For more information about the v1 and v2 formats, see Quota File Formats.

1. By default, Lustre 1.6.5 uses the v2 format for administrative quotas. Previous releases use quota v1.

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:

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 `quotacheck` command scans the entire file system (sub-quotachecks are run on both the MDS and the OSTs) to recompute disk usage (for both inodes and blocks) on a per-UID/GID basis. If there are many files in Lustre, `quotacheck` 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 is not allowed to allocate more space on the file system.

Note – For Lustre 1.6 releases prior to version 1.6.5, and 1.4 releases prior to 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 (such as when a broken Lustre installation or build is used), re-run `quotacheck` after examining the server kernel logs and fixing the root problem.

The `lfs` command now includes these command options to work with quotas:

- `quotaon` — announces to the system that disk quotas should be enabled on one or more file systems. The file system quota files must be present in the root directory of the specified file system.
- `quotaoff` — announces to the system that the specified file systems should have all disk quotas turned off.
- `setquota` — used to specify the quota limits and tune the grace period. By default, the grace period is one week.

Usage:

```
setquota [ -u | -g ] <name> <block-softlimit> <block-hardlimit>
<inode-softlimit> <inode-hardlimit> <filesystem>
setquota -t [ -u | -g ] <block-grace> <inode-grace> <filesystem>
lfs > setquota -u bob 307200 309200 10000 11000 /mnt/lustre
```

In the above example, the quota is set to 300 MB (309200*1024) and the hard limit is 11,000 files on user `bob`. 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.

Quota displays the quota allocated and consumed for each Lustre device. This example shows the result of the previous `setquota`:

```
# lfs quota -u bob /mnt/lustre 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 Resetting the Quota

To reset the quota that was previously established for a user, run:

```
# setquota -u $user 0 0 0 0 /srv/testfs
```

Then run:

```
# setquota -u $user a b c d /srv/testfs
```

Caution – Do not use `# lfs setquota` to reset the previously-established quota.

9.1.4 Quota Allocation

The Linux kernel sets a default quota size of 1 MB. (For a block, the default is 100 MB. For files, the default is 5000.) 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 example again:

```
# lfs quota -u bob /mnt/lustre

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 alloted to user `bob`, which is further disributed 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.

To turn on the quotas for a user and a group, run:

```
$ lfs quotaon -ug /mnt/lustre
```

To turn off the quotas for a user and a group, run:

```
$ lfs quotaoff -ug /mnt/lustre
```

To set the quotas for a user as 1 GB block quota and 10,000 file quota, run:

```
$ lfs setquota -u {username} 0 1000000 0 10000 /mnt/lustre
```

To list the quotas of a user, run:

```
$ lfs quota -u {username} /mnt/lustre
```

To see the grace time for quota, run:

```
$ lfs quota -t –{u|g} {quota user|group} /mnt/lustre
```

9.1.5 Known Issues with Quotas

Using quotas in Lustre can be complex and there are several known issues.

9.1.5.1 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 userA 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 (just like echo XXXX > /proc/fs/lustre/osc/lustre-OST*/max_dirty_mb).

9.1.5.2 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 4TB.
- Lustre versions 1.4.12 and 1.6.5 support quota limits of 4TB and greater in Lustre configurations with OST storage limits of 4TB and less.
- Future Lustre versions are expected to support quota limits of 4TB 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

9.1.5.3 Quota File Formats

Lustre 1.6.5 introduces a new quota file format (v2) 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. In a future Lustre release, the v2 format will be added for operational quotas. A few notes regarding the current quota file formats:

- Lustre 1.6 uses `mdt.quota_type` to force a specific quota version (2 or 1).[2]
  - For the v2 quota file format, (OBJECTS/admin_quotafile_v2.{usr,grp})
  - For the v1 quota file format, (OBJECTS/admin_quotafile.{usr,grp})
- If quotas do not exist or look broken, `quotacheck` creates quota files of a required name and format.
  - If Lustre is using the v2 quota file format, 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 MDS.
- During conversion of a v1 quota file to a v2 quota file, the v2 quota file is marked as broken, to avoid its later usage in case of a crash.
- Quota module refuses to use broken quota files (keeping quota off).

2. Lustre 1.4 uses a quota file dependent on quota32 configuration options.

9.1.6 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 req 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.

9.1.6.1 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 "stats" in /proc/fs/lustre/lquota/ and send them to Lustre Support.**

Note – Proc quota entries are collected in /proc/fs/lustre/obdfilter/lustre-OSTXXXX/quota* and /proc/fs/lustre/mds/lustre-MDTXXXX/quota*, and copied to /proc/fs/lustre/lquota. To maintain compatibility, the old quota proc entries in the /proc/fs/lustre/obdfilter/lustre-OSTXXXX/ and /proc/fs/lustre/mds/lustre-MDTXXXX/ folders are not deleted in the current Lustre release, but they may be deprecated in the future. 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 and OSS

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.

OSS

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 MTF[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 use many smaller OSTs, instead of fewer, large-size OSTs. Following this recommendation will provide you with more IOPS, by having independent RAID sets instead of a single one.

Suggestion: Use RAID 5 with 5 or 9 disks, or RAID 6 with 6 or 10 disks, each on a different controller. Ideally, the RAID configuration should allow 1 MB Lustre RPCs to fit evenly on one RAID stripe without requiring an expensive read-modify-write cycle.

```
<stripe_width> = <chunk_size> * ( <disks> - <parity_disks> ) <=1 MB
```

where *<parity_disks>* is 1 for RAID 5 and 2 for RAID 6. If the RAID configuration does not allow *<chunk_size>* to fit evenly into 1 MB, select *<chunk_size>*, such that *<stripe_width>* is close to 1 MB, but not larger.

For example, RAID 6 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>" ...
```

1. Mean Time to Failure

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 resynchronized, 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

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 to Format MDT and OST File Systems.

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.

10.2 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.

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 Lustre Installation.**

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.

   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 *`<chunk_size>`* <= 1024KB/8. Therefore, the largest valid chunk size is 128KB.

4. These enhancements have mostly improved write performance.

a. Create a RAID array for an OST. On the OSS, run:

```
$ mdadm --create <array_device> -c <chunk_size> -l \
<raid_level> -n <active_disks> -x <spare_disks> <block_devices>
```

where:

<array_device>	RAID array to create, in the form of /dev/md*X*
<chunk_size>	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 be resynced, a necessary process which resynchronizes the various disks in the array so their contents match. The arrays may be used during the resync process (including formatting the OSTs), but performance will not be as high as usual. The resync 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 13

Bonding

This chapter describes how to set up bonding with Lustre, and includes the following sections:

- Network Bonding
- Requirements
- Using Lustre with Multiple NICs versus Bonding NICs
- Bonding Module Parameters
- Setting Up Bonding
- Configuring Lustre with Bonding

13.1 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.)

13.2 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.

13.3 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.

13.4 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.

13.5 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
```

13.5.1 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
```

13.6 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)
```

13.6.1 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 **14**

Upgrading Lustre

The chapter describes how to upgrade and downgrade Lustre versions and includes the following sections:

14.1 Lustre Interoperability

For Lustre 1.6.x, the following upgrades are supported:

- Lustre 1.4.12 (latest 1.4.x version) to the latest Lustre 1.6.x (currently, 1.6.7.1).
- One minor version to the next (for example, 1.6.6 > 1.6.7.1 and 1.4.11 > 1.4.12).

For Lustre 1.6.x, downgrades in the same ranges are supported.

- If you upgrade from Lustre 1.4.12 > 1.6.x (latest version), you can downgrade to version 1.4.12.
- If you upgrade from one minor version to the next (for example Lustre 1.6.6 > 1.6.7.1), you can downgrade to earlier minor version (e.g., version 1.6.6).

Caution – A fresh installation of Lustre 1.6.x is not guaranteed to be downgradable to an earlier Lustre version.

14.2 Upgrading from Lustre 1.4.12 to Latest 1.6.x Version

Use the procedures in this chapter to upgrade Lustre version 1.4.12 to the latest 1.6.x version (for example, 1.4.12 > 1.6.7.1).

Note – In Lustre version 1.6 and later, the file system name (`--fsname` parameter) is limited to 8 characters, so it fits on the disk label.

14.2.1 Prerequisites to Upgrading Lustre

Remember the following points before upgrading Lustre.

- The MDT must be upgraded before the OSTs are upgraded.

1. **Shut down lconf failover.**
2. **Install the new modules.**
3. **Run tunefs.lustre.**
4. **Mount startup.**

- A Lustre upgrade can be done across a failover pair.

1. **On the backup server, install the new modules.**
2. **Shut down lconf failover.**
3. **On the new server, run tunefs.lustre.**
4. **On the new server, mount startup.**
5. **On the primary server, install the new modules.**

14.2.2 Supported Upgrade Paths

The following Lustre upgrade paths are supported.

Entire file system or individual servers / clients

- Servers can undergo a "rolling upgrade", in which individual servers (or their failover partners) and clients are upgraded one at a time and restarted, so that the file system never goes down. This type of upgrade limits your ability to change certain parameters.
- The entire file system can be shut down, and all servers and clients upgraded at once.
- Any combination of the above two paths.

Interoperability between the nodes

This describes the interoperability between clients, OSTs, and MDTs.

Clients

- Old live clients can continue to 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 (use old mount format for old MDT).

OSTs

- New clients/MDTs can continue to communicate with old OSTs.
- New OSTs can only be started after the MGS has been started (typically this means "after the MDT has been upgraded.")

MDTs

- New clients can communicate with old MDTs.
- New co-located MGS/MDTs can be started at any point.
- New non-MGS MDTs can be started after the MGS starts.

Note – The limitation with interoperability is that old clients cannot mount a file system which was created by a new MDT.

Note – If your system is upgraded from 1.4.x to 1.6.x, you can mount the Lustre client on both Lustre versions. If the file system was originallycreated using Lustre 1.6.x, you will not be able to mount the file system created using an earlier Lustre version.

14.2.3 Starting Clients

You can start a new client with an old MDT by using the old format of the client mount command:

```
client# mount -t lustre <mdtnid>:/<mdtname>/client <mountpoint>
```

You can start a new client with an upgraded MDT by using the new format and pointing it at the MGS, not the MDT (for co-located MDT/MGS, this is the same):

```
client# mount -t lustre <mgsnid>:/<fsname> <mountpoint>
```

Old clients always use the old format of the mount command, regardless of whether the MDT has been upgraded or not.

14.2.4 Upgrading a Single File system

`tunefs.lustre` will find the old client log on an 1.4.x MDT that is being upgraded to 1.6. (If the name of the client log is not "client", use the lustre_up14.sh script, described in Step 2 and Step 3.)

1. **Shut down the MDT.**

   ```
   mdt1# lconf --failover --cleanup config.xml
   ```

2. **Install the new Lustre version and run tunefs.lustre to upgrade the configuration.**

   There are two options:

   - Rolling upgrade keeps a copy of the original configuration log, allowing immediate reintegration into a live file system, but prevents OSC parameter and failover NID changes. (The writeconf procedure can be performed later to eliminate these restrictions. For details, see Running the Writeconf Command.)

     ```
     mdt1# tunefs.lustre --mgs --mdt --fsname=testfs /dev/sda1
     ```

   - i.--writeconf begins a new configuration log, allowing permanent modification of all parameters (see Changing Parameters), but requiring all other servers and clients to be stopped at this point. No clients can be started until all OSTs are upgraded.

```
[root@mds1]# tunefs.lustre --mgs --writeconf --mgs --mdt
--fsname=ldiskfs /dev/hda4
checking for existing Lustre data: found CONFIGS/mountdata
Reading CONFIGS/mountdata
Read previous values:
Target:      testfs-MDT0000
Index:       0
UUID:        mds-1_UUID
Lustre FS:   testfs
Mount type:  ldiskfs
Flags:       0x205
(MDT MGS upgrade1.4 )
Persistent mount opts: errors=remount-ro,iopen_nopriv,user_xattr
Parameters:
Permanent disk data:
Target:      ldiskfs-MDT0000
Index:       0
UUID:        mds-1_UUID
Lustre FS:   ldiskfs
Mount type:  ldiskfs
Flags:       0x305
             (MDT MGS writeconf upgrade1.4 )
Persistent mount opts: errors=remount-ro,iopen_nopriv,user_xattr
Parameters:
Writing CONFIGS/mountdata
Copying old logs
```

3. **Start the upgraded MDT.**

```
mdt1# mkdir -p /mnt/test/mdt
mdt1# mount -t lustre /dev/hda4 /mnt/test/mdt
mdt1 # df
Filesystem     1K-blocks Used      Available Use%   Mounted on
/dev/hda2      10080520  4600820   4967632   49%    /
/dev/hda1      101086    14787     81080     16%    /boot
none           501000    0         501000    0%     /dev/shm
/dev/hda4      23339176  455236    21550144  3%     /mnt/test/mdt
```

4. **Upgrade and start the OSTs for the file system in a similar manner, except they need the address of the MGS. Old installations may also need to specify the OST index (for instance, --index=5).**

```
ost1# tunefs.lustre --ost --fsname=lustre --mgsnode=mds /dev/sda4
checking for existing Lustre data: found last_rcvd
tunefs.lustre: Unable to read /tmp/dirQi2cwV/mountdata (No such
file or directory.)
Trying last_rcvd
Reading last_rcvd
Feature compat=2, incompat=0
Read previous values:
Target:
Index:          0
UUID:           ost1_UUID
Lustre FS:      lustre
Mount type:     ldiskfs
Flags:          0x202
                (OST upgrade1.4 )
Persistent mount opts:
Parameters:
Permanent disk data:
Target:         lustre-OST0000
Index:          0
UUID:           ost1_UUID
Lustre FS:      lustre
Mount type:ldiskfs
Flags:          0x202
                (OST upgrade1.4 )
Persistent mount opts: errors=remount-ro,extents,mballoc
Parameters: mgsnode=192.168.10.34@tcp
Writing CONFIGS/mountdata 11.1.5 Upgrading Multiple File Systems
with a Shared MGS
Ost-1# mount -t lustre /dev/sda4 /mnt/test/ost/
Ost1# df
Filesystem     1K-blocks Used      Available Use%   Mounted on
/dev/sda2      10080520  3852036   5716416   41%    /
/dev/sda1      101086    14964     80903     16%    /boot
none           501000    0         501000    0%     /dev/shm
/dev/sda4      101492248 471672    95781780  1%     /mnt/test/ost
```

14.2.5 Upgrading Multiple File Systems with a Shared MGS

The upgrade order is: MGS first, then for any single file system the MDT must be upgraded and mounted, and then the OSTs for that file system. If the MGS is co-located with the MDT, the old configuration logs stored on the MDT are automatically transferred to the MGS. If the MGS is not co-located with the MDT (for a site with multiple file systems), the old config logs must be manually transferred to the MGS.

1. **Format the MGS node, but do not start it.**

```
mgsnode# mkfs.lustre --mgs /dev/hda4
```

2. **Mount the MGS disk as type ldiskfs.**

```
mgsnode# mount -t ldiskfs /dev/hda4 /mnt/mgs
```

3. **For each MDT, copy the MDT and client startup logs from the MDT to the MGS, renaming them as needed. There is a script that helps automate this process—lustre_up14.sh**

```
mdt1# lustre_up14 /dev/hda4 lustre
debugfs 1.35 (28-Feb-2004)
/dev/hda4: catastrophic mode - not reading inode or group bitmaps
Copying log 'mds-1' to 'lustre-MDT0000'. Okay [y/n]?y
Copying log 'client' to 'lustre-client'. Okay [y/n]?y
ls -l /tmp/logs
total 24
-rw-r--r--  1 root root 9448 Oct 22 17:46 lustre-client
-rw-r--r--  1 root root 9080 Oct 22 17:46 lustre-MDT0000
mdt1# cp /tmp/logs/lustre-* /mnt/tmp/CONFIGS/
cp: overwrite `/mnt/tmp/CONFIGS/lustre-client'? y
cp: overwrite `/mnt/tmp/CONFIGS/lustre-MDT0000'? y
```

4. **Unmount the MGS ldiskfs mount.**

```
mgsnode# umount /mnt/mgs
```

5. **Start the MGS.**

```
mgsnode# mount -t lustre /dev/hda4 /mnt/mgs
```

6. **Shut down one of the old MDTs.**

```
mdt1# lconf --failover --cleanup config.xml
```

7. **Upgrade the old MDT.**

```
install new Lustre 1.6
mdt1# tunefs.lustre --mdt --nomgs --fsname=testfs \
--mgsnode=mgsnode@tcp0 /dev/hda4
```

(`--nomgs` is *required* to upgrade a non-co-located MDT).

8. **Start the upgraded MDT.**

```
mdt1# mount -t lustre /dev/hda4 /mnt/test/mdt
```

9. **Upgrade and start OSTs for this file system.**

```
ost1# lconf --failover --cleanup config.xml
install new Lustre 1.6
ost1# tunefs.lustre --ost --fsname=lustre --mgsnode=mgsnode@tcp0 \
/dev/sdc
ost1# mount -t lustre /dev/sdc /mnt/test/ost1
```

10. **Upgrade the other MDTs in a similar manner. Remember:**

- The MGS must not be running (mounted) when the backing disk is mounted as ldiskfs.
- The MGS must be running when first starting a newly-upgraded server (MDT or OST).

14.3 Upgrading Lustre 1.6.x to the Next Minor Version

To upgrade Lustre 1.6.x to the next minor version (for example, Lustre 1.6.6 > 1.6.7.1), perform these steps:

1. **Check the current Lustre version on the MDS.**

```
root@mds# uname -a
root@mds# Linux mds.sun.com 2.6.18-8.1.14.el5_lustre.1.6.4.2smp #1
SMP Wed Jan 16 20:49:25 EST 2008 i686 athlon i386 GNU/Linux
```

2. **Check the name of the file system.**

```
[root@mds ~]# cat /proc/fs/lustre/mgs/MGS/filesystems
sunfs
```

3. **Umount the old file system in this order.**
   - Client
   - OST
   - MDS

```
[root@client ~]# df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda2              6940516   4051136   2531132  62% /
/dev/sda1               101086     14484     81383  16% /boot
tmpfs                   271844         0    271844   0% /dev/shm
mds@tcp0:/sunfs        4128416    291800   3626904   8% /mnt
[root@client ~]# umount /mnt
```

4. **Cross-check the unmount. You must verify the unmount before upgrading the Lustre version.**

```
[root@client ~]# df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda2              6940516   4051136   2531132  62% /
/dev/sda1               101086     14484     81383  16% /boot
tmpfs                   271844         0    271844   0% /dev/shm
```

5. **Unmount the file system on the MDT and all OSTs in a similar manner.**

6. **Install new Lustre version and restart the nodes with new kernel on the MGS and MDT.**

```
[root@mds ~]# tunefs.lustre --mgs --writeconf --mgs --mdt --fsname=
sunfs /dev/sdb
tunefs.lustre --mgs --writeconf --mgs --mdt --fsname=sunfs /dev/sdb
checking for existing Lustre data: found CONFIGS/mountdata
Reading CONFIGS/mountdata
Read previous values:
Target:     sunfs-MDT0000
Index:      0
Lustre FS:  sunfs
Mount type: ldiskfs
Flags:      0x5
              (MDT MGS )
Persistent mount opts: errors=remount-ro,iopen_nopriv,user_xattr
Parameters: mdt.group_upcall=/usr/sbin/l_getgroups
Permanent disk data:
Target:     sunfs-MDT0000
Index:      0
Lustre FS:  sunfs
Mount type: ldiskfs
Flags:      0x105
              (MDT MGS writeconf )
Persistent mount opts: errors=remount-ro,iopen_nopriv,user_xattr
Parameters: mdt.group_upcall=/usr/sbin/l_getgroups
```

7. **Write configurations to mount data.**

```
[root@mds ~]# mount -t lustre /dev/sdb /mnt/data/mdt/
[root@mds ~]# df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda2              6940516   4173316   2408952  64% /
/dev/sda1               101086     14548     81319  16% /boot
tmpfs                   271804         0    271804   0% /dev/shm
/dev/sdb              1834832     90196   1639780   6% /mnt/data/mdt
```

8. **Mount the OST.**

```
oss #  mount -t lustre /dev/sdb /mnt/ost
```

If an error occurs, use this command:

```
oss # tunefs.lustre --ost --fsname=sunfs --mgssnode=mds /dev/sdb
```

9. **After installing the new Lustre modules, mount the file system on the client side.**

```
client # mount -t lustre mds@tcp0:/sunfs /mnt/client
```

14.4 Downgrading from Latest 1.6.x Version to Lustre 1.4.12

This section describes how to downgrade from the latest 1.6.x version to Lustre 1.4.12.

14.4.1 Downgrade Requirements

- The file system must have been upgraded from 1.4.x. In other words, a file system created or reformatted under Lustre 1.6.x cannot be downgraded.
- Any new OSTs that were dynamically added to the file system will be unknown in version 1.4.x. It is possible to add them back using `lconf --write-conf`, but you must be careful to use the correct UUID of the new OSTs.
- Downgrading an MDS that is also acting as an MGS prevents access to all other file systems that the MGS serves.

14.4.2 Downgrading a File System

To downgrade a file system:

1. **Shut down all clients.**
2. **Shut down all servers.**
3. **Install Lustre 1.4.x on the client and server nodes.**
4. **Restart the servers (OSTs, then MDT) and clients.**

Caution – When you downgrade Lustre, all OST additions and parameter changes made since the file system was upgraded are lost.

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:

- Installing the Lustre SNMP Module
- Building the Lustre SNMP Module
- Using the Lustre SNMP Module

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

This chapter describes how to perform backup and restore on Lustre, and includes the following sections:

15.1 Lustre Backups

Lustre provides backups at several levels. Generally, file system-level backups are recommended over device-level backups.

15.1.1 File System-level Backups

File system-level backups give you full control over the files to back up, and allow 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 directory, filesets for a single user, files incremented by date, and so on.

15.1.2 Device-level Backups

Full, device-level backups of the MDS and OSTs should be done before replacing hardware, performing maintenance, etc. A device-level backup of the MDS is especially important because, if it fails permanently, the entire file system would need to be restored.

In case of hardware replacement, if the spare storage device is available, then it is possible to take 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 there are problems while reading the data on the original device due to hardware errors, then run the following command to read the data and skip sections with errors.

```
dd if=/dev/{original} of=/dev/{new} bs=4k conv=sync,noerror
```

In spite of hardware errors, the ext3 file system is very robust and it may be possible to recover the file system data after running e2fsck on the new device.

15.1.3 Performing File-level Backups

In some situations, you may want to back up data from a single file on the MDS or an OST file system, rather than back up the entire device. This may be a preferred backup strategy if the storage device is large but has relatively little data, parameter configurations on the ext3 file system need to be changed, or to use less space for backup.

You can mount the ext3 file system directly from the storage device and do a file-level backup. However you **MUST STOP** Lustre on that node.

To do this, back up the Extended Attributes (EAs)[1] stored in the file system. As the current backup tools do not properly save this data, perform the following procedure.

1. Lustre uses EAs to store striping information (location of file data on OSTs).

15.1.3.1 Backing Up an MDS File

To back up a file on the MDS:

1. **Make a mount point for the file system "mkdir /mnt/mds" and mount the file system at that location.**
   - For 2.4 kernels, run:

     ```
     mount -t ext3 {dev} /mnt/mds
     ```
   - For 2.6 kernels, run:

     ```
     mount -t ldiskfs {dev} /mnt/mds
     ```
2. **Change to the mount point being backed up "cd /mnt/mds"**
3. **Back up the EAs, run:**

   ```
   getfattr -R -d -m '.*' -P . > ea.bak
   ```

Note – The `getfattr` command is part of the "attr" package in most distributions. 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.

---

4. **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. It should have an item for each file like:**

   ```
   # file: ROOT/mds_md5sum3.txt
   trusted.lov=
   0s0AvRCwEAAABXoKUCAAAAAAAAAAAAAAAAAAAQAAEAAADD5QoAAAAAAAAAAAAAAAAA
   AAAAAAEAAAA=
   ```
5. **Back up all file system data, run:**

   ```
   tar czvf {backup file}.tgz
   ```
6. **Change directory out of the mounted file system, run:**

   ```
   cd -
   ```
7. **Unmount the file system, run:**

   ```
   umount /mnt/mds
   ```

15.1.3.2 Backing Up an OST File

Follow the same procedure as Backing Up an MDS File (except skip Step 4) and, for each OST device file system, replace `mds` with `ost` in the commands.

15.2 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 device. To get the optimal ext3 parameters, run:**

```
$ mkfs.lustre --fsname {fsname} --reformat --mgs|mdt|ost /dev/sda
```

Caution – Only reformat the node which is being restored. If there are multiple services on the node, do not perform this step as it can cause all devices on the node to be reformatted. In that situation, follow these steps:

For MDS file systems, run:

```
mke2fs -j -J size=400 -I {inode_size} -i 4096 {dev}
```

where `{inode_size}` is at least 512 and possibly larger if the default stripe count is **> 10 (inode_size = power_of_2_>=_than(384 + stripe_count * 24))**[2]

For OST file systems, run:

```
mke2fs -j -J size=400 -I 256 -i 16384 {dev}"
```

---

2. **Enable ext3 file system directory indexing.**

```
tune2fs -O dir_index {dev}
```

2. In the mke2fs command, the -I option is the size of the inode and the -i option is the ratio of inodes to space in the file system: inode_count = device_size / inode_ratio. Set the -i option to 4096 so Extended Attributes (EAs) can fit on the inode as well. Otherwise, you have to make an indirect allocation to hold the EAs, which impacts performance owing to the additional seeks.

3. **Mount the file system.**
   - For 2.4 kernels, run:

     ```
     mount -t ext3 {dev} /mnt/mds
     ```
   - For 2.6 kernels, run:

     ```
     mount -t ldiskfs {dev} /mnt/mds
     ```

4. **Change to the new file system mount point, run:**

   ```
   cd /mnt/mds
   ```

5. **Restore the file system backup, run:**

   ```
   tar xzvpf {backup file}
   ```

6. **Restore the file system EAs, run:**

   ```
   setfattr --restore=ea.bak (not required for OST devices)
   ```

7. **Remove the recovery logs (now invalid), run:**

   ```
   rm OBJECTS/* CATALOGS
   ```

Note – If the file system is in use during the restore process, run the `lfsck` tool (part of e2fsprogs) to ensure that the file system is coherent.

It is not necessary to run this tool if the backup of all device file systems occurs at the same time after stopping the entire Lustre file system. After completing the file system should be immediately usable without running lfsck. There may be few I/O errors reading from files that are present on the MDS, but not on the OSTs. However, the files that are created after the MDS backup are not visible or accessible.

15.3 LVM Snapshots on Lustre Target Disks

Another disk-based backup option is to leverage the Linux LVM snapshot mechanism to maintain multiple incremental backups of a Lustre file system. But LVM snapshots cost CPU cycles as new files are written, so taking snapshots of the main Lustre file system will probably result in unacceptable performance losses.

To get around this problem, create a new, backup file system and periodically back up new/changed files. Take periodic snapshots of this backup file system to create a series of compact "full" backups.

15.3.1 Creating LVM-based Lustre File System As a Backup

To create an LVM-based backup Lustre file system.

1. **Create LVM volumes for the MDT and OSTs.**

   First, create LVM devices for your MDT and OST targets. Do not use the entire disk for the targets, as some space is required for the snapshots. The snapshots size start out as 0, but they increase in size as you make changes to the backup file system. In general, if you expect to change 20% of your file system between backups, then the most recent snapshot will be 20% of your target size, the next older one will be 40%, and so on.

```
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 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
```

15.3.2 Backing Up New Files to the Backup File System

This is your nightly backups of your real on-line Lustre file system.

```
cfs21:~# cp /etc/passwd /mnt/main
cfs21:~# cp /etc/fstab /mnt/main
cfs21:~# ls /mnt/main
 fstab  passwd
```

15.3.3 Creating LVM Snapshot Volumes

Whenever you want to make a "checkpoint" of your Lustre file system, you create LVM snapshots of all the target disks in "main". You must decide the maximum size of a snapshot ahead of time,however you can dynamically change this later. The size of a daily snapshot is dependent on the amount of data you change daily in your on-line file system. It is also 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 your volume group. You can also dynamically add disks to the volume group if needed.

The snapshots of the target disks (MDT, OSTs) should be taken at the same point in time, making sure that cronjob updating "main" is not running, since that is the only job writing to the disks.

```
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
```

15.3.4 Restoring From Old Snapshot

1. **Rename the snapshot**

   Rename the snapshot file system from "main" to "back" so that you can mount it without unmounting "main". This is not a requirement. Use the `--reformat` flag to `tunefs.lustre` to force the name change.

```
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 snapshot file system**

```
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
```

Note the old directory contents, as of the snapshot time.

```
cfs21:~/cfs/b1_5/lustre/utils# ls /mnt/back
 fstab  passwds
```

15.3.5 Delete Old Snapshots

To reclaim disk space, you can erase old snapshots, as your backup policy dictates.

```
lvremove /dev/volgroup/MDTb1
```

\You can also extend or shrink snapshot volumes if you find your daily deltas are smaller or larger than you had planned for.

```
lvextend -L10G /dev/volgroup/MDTb1
```

CHAPTER **16**

POSIX

This chapter describes POSIX and includes the following sections:

- Installing POSIX
- Running POSIX Tests Against Lustre
- Isolating and Debugging Failures

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. It is not a standard for clusters.

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 can not see the metadata. POSIX offers strict mandatory locking which gives guarantee of semantics. Users do not have control on these locks.

The current Lustre POSIX is comparable with NFS. Lustre 1.8 promises strong security with features like GSS/Kerberos 5. This enables graceful handling of users from multiple realms which, in turn, introduce multiple UID and GID databases.

Note – Although used mainly with UNIX systems, the POSIX standard can apply to any operating system.

16.1 Installing POSIX

To install POSIX (used for testing Lustre):

1. **Download all POSIX files from:**

   http://downloads.clusterfs.com/public/tools/benchmarks/posix/

   - `lts_vsx-pcts-1.0.1.2.tgz`
   - `install.sh`
   - `myscen.bld`
   - `myscen.exec`

Caution – Do **not** configure or mount a Lustre file system yet.

---

2. **Run the install.sh script and select /home/tet for the root directory for the test suite installation.**

3. **Install users and groups. Accept the defaults for the packages to be installed.**

4. **To avoid a bug in the installation scripts where the test directory is not created properly, create a temporary directory to hold the POSIX tests when they are built.**

```
$ mkdir -p /mnt/lustre/TESTROOT;chown vsx0.vsxg0
```

5. **Log in as the test user.**

```
su - vsx0
```

6. **Build the test suite, run:**

```
../setup.sh
```

   Most of the defaults are correct, except the root directory from which to run the test sets. For this setting, specify `/mnt/lustre/TESTROOT`. Do NOT install pseudo languages.

7. **When the system displays this prompt:**

```
Install scripts into TESTROOT/BIN..?
```

Do not immediately respond. Using another terminal (as stopping the script does not work), replace the files /home/tet/test_sets/scen.exec and /home/tet/test_sets/scen.bld with myscen.exec and myscen.bld (downloaded earlier).

```
$ cp .../myscen.bld  /home/tet/test_sets/scen.bld
$ cp .../myscen.exec /home/tet/test_sets/scen.exec
```

This limits the tests run only to the relevant file systems and avoids additional hours of other tests on sockets, math, stdio, libc, shell, and so on.

8. **Continue with the installation.**

   a. **Build the test sets.**

   It proceeds to build and install all of the file system tests.

   b. **Run the test sets.**

   Even though it is running them on a local file system, this is a valuable baseline to compare with the behavior of Lustre. It should put the results into /home/tet/test_sets/results/0002e/journal. Rename or symlink this directory to /home/tet/test_sets/results/ext3/journal (or to the name of the local file system on which the test was run).

   Running the full test takes about five minutes. Do not re-run any failed test. Results are in a lengthy table at /home/tet/test_sets/results/report.

9. **Save the test suite to run further tests on a Lustre file system. Tar up the tests, so that you do not have to rebuild each time.**

16.2 Running POSIX Tests Against Lustre

To run the POSIX tests against Lustre:

1. **As root, set up your Lustre file system, mounted on /mnt/lustre (for instance, sh llmount.sh) and untar the POSIX tests back to their home.**

   ```
   $ tar --same-owner -xzpvf /path/to/tarball/TESTROOT.tgz -C \
   /mnt/lustre
   ```

   As the vsx0 user, you can re-run the tests as many times as you want. If you are newly logged in as the vsx0 user, you need to source the environment with '. profile' so that your path and environment is set up correctly.

2. **Run the POSIX tests, run:**

   ```
   $ . /home/tet/profile
   $ tcc -e -s scen.exec -a /mnt/lustre/TESTROOT -p
   ```

   New results are placed in new directories at `/home/tet/test_sets/results` Each result is given a directory name similar to 0004e (an incrementing number which ends with `e` (for test execution) or `b` (for building tests).

3. **To look at a formatted report, run:**

   ```
   $ 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 as it helps to find test failures that are specific to the file system, instead of the Linux VFS or kernel. Up to six test results can be compared at one time. It is often useful to rename the results directory to have more interesting names so that they are meaningful in the future.

16.3 Isolating and Debugging Failures

In the case of Lustre failures, you need to capture information about what is happening at runtime. For example some tests may cause kernel panics, depending on your Lustre configuration. By default, debugging is not enabled in the POSIX test suite. You need to turn on the VSX debugging options. There are two debug options of note in the config file tetexec.cfg, 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 can 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 is not useful in case of kernel panic and Lustre (or your machine) crashes.

VSX_DBUG_FLAGS=xxxxx

The following example makes VSX 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 also have TET print out verbose debug messages by inserting the `-T` option when running the tests. For example:

```
$  tcc -Tall5 -e -s scen.exec -a /mnt/lustre/TESTROOT -p 2>&1 | tee
/tmp/POSIX-command-line-output.log
```

VSX prints out detailed messages in the report for failed tests. This includes the test strategy, operations done by the test suite, and the failures. Each subtest (for instance, 'access', 'create') usually contains many single tests. The report shows exactly which single testing 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 narrow 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, with an appropriate name (like `TESTROOT-chmod-5-only.tgz`) and re-run the POSIX suite. It may also be helpful to edit the `scen.exec` file to run only test set in question.

```
 "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. You may have to substitute the edited source files into the source tree (following the installation 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, similar to `tcc -p -b -s $HOME/scen.bld $*` but it does not work outside the script.

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 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 can be run automatically or manually against a local disk. Obdfilter-survey profiles the overall throughput of storage hardware[1], by sending ranges of workloads to the OSTs (that vary 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 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=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

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

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 is a configuration of a test case, which defines individual point-to-pointer network conversation all running in parallel. A user can specify test properties, such as RDMA operation type, source group, target group, distribution of test nodes, concurrency of test, etc.

18.4.1.7 Batch

A test batch is a named collection of tests. All tests in a batch 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.

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 Concepts

This section describes the fundamental concepts of LNET self-test.

18.4.3 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.

18.4.3.1 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
```

18.4.3.2 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.

18.4.3.3 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
```

18.4.3.4 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:

Lustre offers substantial recovery support to deal with node or network failure, and returns the cluster to a reliable, functional state. When Lustre is in recovery mode, it means that the servers (MDS/OSS), judge there is a stop of file system in an unclean state. In other words, unsaved data may be in the client cache. To save this data, the file system re-starts in recovery mode and makes the clients write the data to disk.

19.1 Recovering Lustre

In Lustre recovery mode, the servers attempt to contact all clients and request they replay their transactions.

If all clients are contacted and they are recoverable (they have not rebooted), then recovery proceeds and the file system comes back with the cached client-side data safely saved to disk.

If one or more clients are not able to reconnect (due to hardware failures or client reboots), then the recovery process times out, which causes all clients to be expelled. In this case, if there is any unsaved data in the client cache, it is not saved to disk and is lost. This is an unfortunate side effect of allowing Lustre to keep data consistent on disk.

19.2 Types of Failure

Different types of failure can cause Lustre to enter recovery mode:

Client (compute node) failure

- MDS failure (and failover)
- OST failure
- Transient network partition
- Network failure
- Disk state loss
- Down node
- Disk state of multiple, out-of-sync systems

Currently, all failure and recovery operations are based on the notion of connection failure. All imports or exports associated with a given connection are considered as failed if any of them do.

19.2.1 Client Failure

Lustre supports for recovery from client failure based on the revocation of locks 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 or a bulk data operation times out, the system removes the client from the cluster. This action allows other clients to acquire locks blocked by the dead client, and it also frees resources (such as file handles and export data) associated with the client. This scenario can be caused by a client node system failure or a network partition.

19.2.2 MDS Failure (and Failover)

Reliable 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 launches the MetadataReplay function. MetadataReplay ensures that the replacement MDS re-accumulates the state resulting from transactions whose effects were visible to clients, but which were not committed to disk. Transaction numbers ensure that the operations replay occurs in the same order as the original integration. Additionally, 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. During the recovery window, only clients that were connected at the time of MDS failure are permitted to reconnect.

ClientUpcall, a user-space policy program, manages the re-connection to a new or rebooted MDS. ClientUpcall is responsible to set up necessary portals, routes and connections, and indicates which connection UUID should replace the failed one.

19.2.3 OST Failure

When an OST fails or is severed from the client, Lustre marks the corresponding OSC as inactive, and the LOV avoids making stripes for new files on that OST. Operations that operate on the "whole file", such as determining file size or unlinking, skips inactive OSCs (and OSCs that become inactive during the operation). Attempts to read from or write to an inactive stripe result in an -EIO error being returned to the client.

As with the MDS failover case, Lustre invokes the ClientUpcall when it detects an OST failure. If and when the upcall indicates that the OST is functioning again, Lustre reactivates an OSC in question and makes file data from stripes on the newly-returned OST available for reading and writing.

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.

19.2.4 Network Partition

The partition can be transient. Lustre recovery occurs in following sequence:

- Clients can detect "harmless partition" upon reconnecting. Dropped-reply cases require ReplyReconstruction.
- Servers evict clients.
- ClientUpcall may try other routers. The arbitrary configuration change is possible the message 'Failed Recovery - ENOTCONN' is given for evicted clients.
- Process invalidates all entries and locks. Eventually, the file system finishes recovering and returns to normal operation. You may check the progress of Lustre recovery by looking at the recovery_status proc entry for each device on the OSSs, for example: `cat /proc/fs/lustre/obdfilter/ost1/recovery_status`
- The file system may get stuck in recovery if any servers are down or if any servers have thrown a Lustre bug (LBUG); check `/proc/fs/lustre/health_check`.

PART III Lustre Tuning, Monitoring and Troubleshooting

The part includes chapters describing how to tune, debutg 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`).

20.1.0.1 OSS Service Thread Count

The `oss_num_threads` parameter allows the number of OST service threads to be specified at module load time on the OSS nodes:

```
options ost oss_num_threads={N}
```

An OSS can have a maximum of 512 service threads and a minimum of 2 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

In such cases, a larger 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 (ost_num_threads module parameter to ost.ko module).

Determining the optimum number of OST threads is a process of trial and error. You may want to start with a number of OST threads equal to the number of actual disk spindles on the node. If you use RAID5+, subtract any dead spindles not used for actual data (e.g., 1/3 of spindles for RAID5), and monitor the performance of clients during usual workloads. If performance is degraded, increase the thread count and see how that works until performance is degraded again or you reach satisfactory performance.[1]

1. If your disk configuration does not have writeback cache enabled and your activity is mostly writes, consider trying the patch in bug 16919 (Bugzilla). It removes synchronous journal commit requirements, and should speed up OST writes (unless you already use fast external journal or writeback cache is enabled that mitigates the synchronousness of the journal commit).

20.1.1 MDS Threads

There is a similar parameter for the number of MDS service threads:

```
options mds mds_num_threads={N}
```

At this time, we have not tested to determine the optimal number of MDS threads. 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 loading within a factor of 4. The default is calculated the same way as before (as explained in OSS Service Thread Count).

Setting the `*_num_threads module` parameter disables the automatic thread creation behavior.

20.1.1.1 I/O Scheduler

Select the best I/O scheduler for your setup.

Try different I/O schedulers (kernel parameter elevator=on old kernels or echo <scheduler> >/sys/block/<dev>/queue/scheduler), because they behave differently depending on storage and load. Benchmark all I/O schedulers and select the best one for your setup. 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 enable_irq_affinity

By default, this parameter is OFF. 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 10 GB Ethernet and more than 2 CPUs, you may see improved performance by turning this parameter to OFF. You should, as always, test to compare the performance impact.

20.3 Options to Format MDT and OST File Systems

The backing file systems on an MDT and OSTs are independent of one another, so the formatting parameters for them should not be same. Sizing the MDT depends solely on how many inodes you want in the entire Lustre file system. This 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 default stripe count option `-c`; but this may change on a per-file basis. 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. Sun recommends 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.

20.3.3 Overriding Default Formatting Options

To override the default formatting options for any of the Lustre backing file systems, 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 file systems, see the mke2fs(8) man page; this section only discusses several Lustre-specific options.

20.3.3.1 Number of Inodes for MDT

To override the inode ratio, use the option `-i <bytes per inode>` (for example) `--mkfsoptions=-i 4096` to create 1 inode per 4096 bytes of file system space).

Note – Use this ratio to make sure that Extended Attributes (EAs) can fit on the inode as well. Otherwise, you have to make an indirect allocation to hold the EAs, which impacts performance owing to the additional seeks.

Alternately, if you are specifying some absolute number of inodes, use the `-N<number of inodes>` option. To avoid unintentional mistakes, do not specify the `-i` option with an inode ratio below one inode per 1024 bytes. Use the `-N` option instead.

By default, a 2 TB MDT has 512M inodes. Currently, the largest supported file system size is 8 TB, which holds 2B inodes. With an MDT inode ratio of 1024 bytes per inode, a 2 TB MDT holds 2B inodes and a 4 TB MDT holds 4B inodes (the maximum number of inodes currently supported by ext3).

20.3.3.2 Inode Size for 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. Lustre (or more specifically the backing ext3 file system), also needs sufficient space for other metadata, like the journal (up to 400 MB), bitmaps and directories. There are also a few regular files that Lustre uses to maintain cluster consistency.

To specify a larger inode size, use the `-I<inodesize>` option. We do NOT recommend specifying a smaller-than-default inode size, as this can lead to serious performance problems; you cannot change this parameter after formatting the file system. The inode ratio must always be larger than the inode size.

20.3.3.3 Number of Inodes for OST

For OST file systems, it is normally advantageous to take local file system usage into account. Try to minimize the number of inodes created on each OST. This helps reduce the format and e2fsck time, and makes more space available for data.

Presently, Lustre has 1 inode per 16 KB of space in the OST file system (by default). In many environments, this is far too many inodes for the average file size. As a general guideline, the 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.

20.4 Network Tuning

During IOR runs, especially reads, one or more nodes may become CPU-bound (which may slow down the remaining nodes and compromise read rates). This issue is likely related to RX overflow errors on the nodes (caused by an upstream e1000 driver). To resolve this issue, increase the RX ring buffer size (default is 256). Use either:

- /sbin/ethtool -G ethX rx 4096
- e1000 module option 'RxDescriptors=4096'

20.5 DDN Tuning

This section provides guidelines to configure DDN storage arrays for use with Lustre. For more complete information on DDN tuning, refer to the performance management section of the DDN manual of your product, available at:

http://www.ddnsupport.com/manuals.html

This section covers the following DDN arrays:

- S2A 8500
- S2A 9500
- S2A 9550

20.5.1 Setting Readahead and MF

For the S2A DDN 8500 storage array, we recommend that you disable the readahead. In a 1000-client system, if each client has up to 8 read RPCs in flight, then this is 8 * 1000 * 1 MB = 8 GB of reads in flight. With a DDN cache in the range of 2 to 5 GB (depending on the model), it is unlikely that the LUN-based readahead would have ANY cache hits even if the file data were contiguous on disk (generally, file data is not contiguous). The Multiplication Factor (MF) also influences the readahead; you should disable it.

CLI commands for the DDN are:

```
cache prefetch=0
cache MF=off
```

For the S2A 9500 and S2A 9550 DDN storage arrays, we recommend that you use the above commands to disable readahead.

20.5.2 Setting Segment Size

The cache segment size noticeably affects I/O performance. Set the cache segment size differently on the MDT (which does small, random I/O) and on the OST (which does large, contiguous I/O). In customer testing, we have found the optimal values to be 64 KB for the MDT and 1 MB for the OST.

Note – The cache size parameter is common to all LUNs on a single DDN and cannot be changed on a per-LUN basis.

These are CLI commands for the DDN.

- For the MDT LUN:

```
$ cache size=64
size is in KB, 64, 128, 256, 512, 1024, and 2048. Default 128
```

- For the OST LUN:

```
$ cache size=1024
```

20.5.3 Setting Write-Back Cache

Performance is noticeably improved by running Lustre with write-back cache turned on. However, there is a risk that when the DDN controller crashes you need to run e2fsck. Still, it takes less time than the performance hit from running with the write-back cache turned off.

For increased data security and in failover configurations, you may prefer to run with write-back cache off. However, you might experience performance problems with the small writes during journal flush. In this mode, it is highly beneficial to increase the number of OST service threads options `ost ost_num_threads=512` in `/etc/modprobe.conf`. The OST should have enough RAM (about 1.5 MB /thread is preallocated for I/O buffers). Having more I/O threads allows you to have more I/O requests in flight, waiting for the disk to complete the synchronous write.

You have to decide whether performance is more important than the slight risk of data loss and downtime in case of a hardware/software problem on the DDN.

Note – There is no risk from an OSS/MDS node crashing, only if the DDN itself fails.

20.5.4 Setting maxcmds

For S2A DDN 8500 array, changing maxcmds to 4 (from the default 2) improved write performance by as much as 30% in a particular case. This only works with SATA-based disks and when only one controller of the pair is actually accessing the shared LUNs.

However, this setting comes with a warning. DDN support does not recommend changing this setting from the default. By increasing the value to 5, the same setup experienced some serious problems.

The CLI command for the DDN client is provided below (default value is 2).

```
$ diskmaxcmds=3
```

For S2A DDN 9500/9550 hardware, you can safely change the default from 6 to 16. Although the maximum value is 32, values higher than 16 are not currently recommended by DDN support.

20.5.5 Further Tuning Tips

Here are some tips we have drawn from testing at a large installation:

- Use the full device instead of a partition (sda vs sda1). When using the full device, Lustre writes nicely-aligned 1 MB chunks to disk. Partitioning the disk can destroy this alignment and will noticeably impact performance.
- Separate the ext3 OST into two LUNs, a small LUN for the ext3 journal and a big one for the "data".
- Since Lustre 1.0.4, we supply ext3 mkfs options when we create the OST like `-j, -J` and so on in the following manner (where `/dev/sdj` has been formatted before as a journal). The journal size should not be larger than 1 GB (262144 4 KB blocks) as it can consume up to this amount of RAM on the OSS node per OST.

  ```
  $ mke2fs -O journal_dev -b 4096 /dev/sdj [optional size]
  ```

Tip – A very important tip—on the S2A DDN 8500 storage array, you need to create one OST per TIER, especially in write through (see output below). This is of concern if you have 16 tiers. Create 16 OSTs consisting of one tier each, instead of eight made of two tiers each.

- Performance is significantly better on the S2A DDN 9500 and 9550 storage arrays with two tiers per LUN.

- Do NOT partition the DDN LUNs, as this causes all I/O to the LUNs to be misaligned by 512 bytes. The DDN RAID stripes and cachelines are aligned on 1 MB boundaries. Having the partition table on the LUN causes all 1 MB writes to do a read-modify-write on an extra chunk, and ALL 1 MB reads to, instead, read 2 MB from disk into the cache, causing a noticeable performance loss.

- You are not obliged to lock in cache the small LUNs.

- Configure the MDT on a separate volume that is configured as RAID 1+0. This reduces the MDT I/O and doubles the seek speed.

For example, one OST per tier:

```
LUNLabel   Owner  Status        Capacity  Block  Tiers  Tier   list
                                (Mbytes)  Size
-------------------------------------------------------------------
0          1      Ready         102400    512    1      1
1          1      Ready         102400    512    1      2
2          1      Ready         102400    512    1      3
3          1      Ready         102400    512    1      4
4          2      Ready [GHS]   102400    4096   1      5
5          2      Ready [GHS]   102400    4096   1      6
6          2      Critical      102400    512    1      7
7          2      Critical      102400    4096   1      8
10         1      Cache Locked  64        512    1      1
11         1      Ready         64        512    1      2
12         1      Cache Locked  64        512    1      3
13         1      Cache Locked  64        512    1      4
14         2      Ready [GHS]   64        512    1      5
15         2      Ready [GHS]   64        512    1      6
16         2      Ready [GHS]   64        4096   1      7
17         2      Ready [GHS]   64        4096   1      8
System verify extent: 16 Mbytes
System verify delay: 30
```

20.6 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.

20.6.1 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.

20.7 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.

20.8 Data Checksums

To avoid the risk of data corruption on the network, a Lustre client can perform end-to-end data checksums[2]. Be aware that at high data rates, checksumming can impact Lustre performance.

2. 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**

Lustre Monitoring and Troubleshooting

This chapter provides information to troubleshoot Lustre, submit a Lustre bug, and Lustre performand tips. It includes the following sections:

21.1 Monitoring Lustre

Several tools are available to monitor a Lustre cluster:

Lustre Monitoring Tool

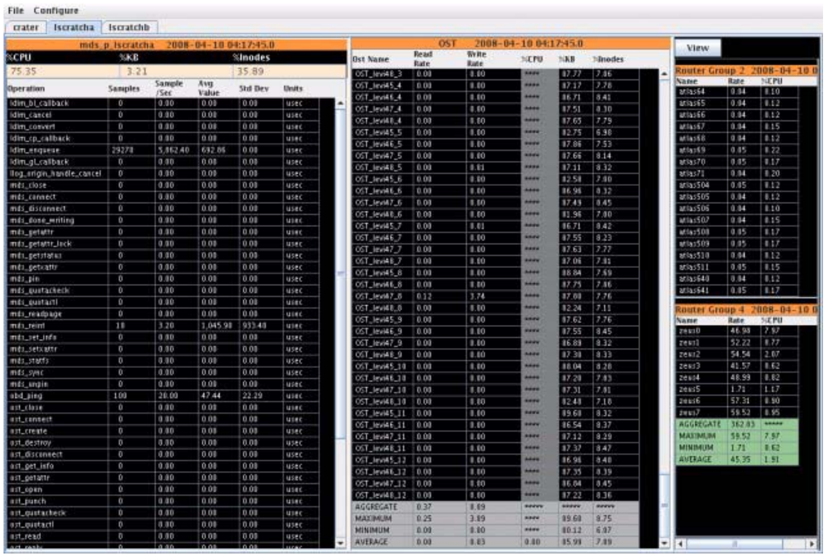
The Lustre Monitoring Tool (LMT[1]) is a Python-based, distributed system that provides a "top" like display of activity on server-side nodes[2] (MDS, OSS and portals routers) on one or more Lustre file systems.

LMT provides a Java-based GUI that reports data for each file system. A tab is presented for each Lustre file system that is being monitored. Within each tab, there are panes presenting the server-side node information (MDS, OSS or portals routers). There is also a tab that presents a multi-level outline view of the sub-components of

1. LMT was developed by Lawrence Livermore National Lab (LLNL) and continues to be maintained by LLNL.
2. Lustre client monitoring is not supported.

each file system component. Data is displayed for OSTs and file systems. For each OST, the current read rate/write rate (in MB/s), % CPU, and % full are displayed. For each file system basis, aggregate MB/s is shown. This is a sample LMT screen:

FIGURE 21-1 LMT sample screen



For more information on LMT, including the setup procedure, see:

http://sourceforge.net/projects/lmt

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.

3. The Lustre Group has made several scripts available for MDS failover testing.

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

Other Monitoring Options

Another option is to script a simple monitoring solution which looks at various reports from ifconfig, as well as the procfs files generated by Lustre.

21.2 Troubleshooting Lustre

Several resouces are available to help use troublshoot Lustre. This section describes error numbers, error messages and logs.

21.2.1 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.

21.2.2 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.

21.2.3 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.

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.

21.3 Submitting a Lustre Bug

If, after troubleshooting your Lustre system, you cannot resolve the problem, consider submitting 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.

21.4 Common Lustre Problems and Performance Tips

This section describes common issues encountered with Lustre, as well as tips to improve Lustre performance.

21.4.1 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 – In Lustre 1.6 and earlier releases, the success of the recovery process is limited by uncommitted client requests that are unable to be replayed. Because clients attempt to replay their requests to the OST and MDT in serial order, a client that cannot replay its requsts causes the recovery stream to stop, and leaves the remaining clients without an opportunity to reconnect and replay their requests.

Lustre 1.8 will introduce the version-based recovery (VBR) feature, which will enable a failed client to be "skipped", so the remaining clients can replay their requests, resulting in a more successful recovery from a downed OST.

21.4.2 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.

21.4.3 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.

21.4.4 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.

21.4.5 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 Lustre 1.6 you can mount a file system with a missing OST.

21.4.6 Changing Parameters

You can set the following parameters at the mkfs time, on a non-running target disk, via `tunefs.lustre` or via a live MGS using the `lctl` command.

With mkfs.lustre

While you are using the `mkfs` command and creating the file system, you can add parameters with the `--param` option:

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

With tunefs.lustre

If a server is stopped, you can add the parameters via `tunefs.lustre` with the same `--param` option:

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

With tunefs.lustre, parameters are "additive" -- to erase all old parameters and just use the newly-specified parameters, use `tunefs.lustre --erase-params --param=....`

With lctl

While a server is running, you can change many parameters via `lctl conf_param`:

```
$ mgs> lctl conf_param testfs-MDT0000.sys.timeout=40
$ anynode> cat /proc/sys/lustre/timeout
```

Note – The `lctl conf_param` command permanently sets parameters in the file system configuration.

21.4.7 Viewing Parameters

To view the parameters set in the configuration log:

1. **Unmount the MGS.**
2. **Mount the MGS disk as**

```
# mount -t ldiskfs /dev/sda /mnt/mgs
```

3. **Use the llog-reader utility to display the contents of the various configuration logs under the CONFIGS directory.**

```
# /usr/sbin/llog-reader /mnt/mgs/CONFIGS/testfs-client
```

4. **Look for items marked "param".**
5. **Check the other logs for parameters that affect those targets, e.g. testfs-MDT0000 for MDT settings.**

The current settings allow you to easily change a parameter. However, there is no simple way to delete a parameter. Shut down all targets and enter the `--writeconf` command to regenerate the logs. Then, add back all of your modified settings.

When you enter the `--writeconf` command, you can set modified settings for each device using following commands.

```
mdt# tunefs.lustre --writeconf --param="failover.mode=failout" /dev/sda
ost1# tunefs.lustre --writeconf  --erase-params
--param="failover.node=192.168.0.13@tcp0"
--param="osc.max_dirty_mb=29.15" /dev/sda
```

Use the `--erase-params` flag to clear old paramers from the tunefs list. Without the `--writeconf` command, clearing the parameters has no effect.

If you change the parameters exclusively via tunefs (<u>not</u> using lctl), then `tunefs.lustre --print` shows you the list of parameters. For production systems, this is the preferred way to set parameters, as the parameters sustain `--writeconf`. Parameters set via `lctl conf_param` do not sustain `--writeconf`.

You can also perform similar operations without unmounting the file system. Run:

```
debugfs -c -R 'dump CONFIGS/testfs-client /tmp/testfs-client' /dev/sda
llog_reader /tmp/testfs-client
```

21.4.8 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*
```

21.4.9 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.

21.4.10 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.

21.4.11 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.

21.4.12 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.

21.4.13 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
  ```

21.4.14 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.

21.4.15 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]
```

21.4.16 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)
```

21.4.17 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.

21.4.18 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.

21.4.19 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.

21.4.20 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 precreation. 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.

21.4.21 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's memory and OSS memory requirements, see Memory Requirements.

21.4.22 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.

21.4.23 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 it is less than 1024, set it to 1024 to improve the performance. If changing this setting does not change the I/O size as reported by Lustre, you may want to examine the SCSI driver code.

CHAPTER 22

LustreProc

This chapter describes Lustre /proc entries and includes the following sections:

- /proc Entries for Lustre
- Lustre I/O Tunables
- Debug Support

The proc file system acts as an interface to internal data structures in the kernel. It can be used to obtain information about the system and to change certain kernel parameters at runtime (sysctl).

The Lustre file system provides several proc file system variables that control aspects of Lustre performance and provide information.

The proc variables are classified based on the subsystem they affect.

22.1 /proc Entries for Lustre

This section describes /proc entries for Lustre.

22.1.1 Finding Lustre

Use the proc files on the MGS to see 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
```

22.1.2 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.

22.1.3 Adaptive Timeouts in Lustre

Lustre 1.6.5 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.

Caution – In Lustre 1.6.5, adaptive timeouts are disabled, by default, in order not to require users applying this maintenance release to use adaptive timeouts. Adaptive timeouts will be enabled, by default, in Lustre 1.8.

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.

22.1.3.1 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. Keep in mind that in most situations, the default values will be usable.

The following parameters can be set as module parameters in modprobe.conf or at runtime in /sys/module/ptlrpc.[1]

Note – This directory path may be different on some 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, 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). In Lustre 1.6.5, the default value is 0*. This setting causes adaptive timeouts to be disabled and the old fixed-timeout method (obd_timeout) to be used. 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 it should time out. **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.

1. The specific sub-directory in ptlrpc containing the parameters is system dependent.

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 (up to its estimate). If the time expires, the Lustre client will enter recovery status and reconnect 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).

* In future releases, the default will be 600 (adaptive timeouts will be enabled).

† 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 1.6.5, adaptive timeouts are disabled, by default.[2] To enable adaptive timeouts, do one of the following:

- At compile time, rebuild Lustre with:

  ```
  --enable-adaptive-timeouts
  ```

- At run time, set `at_max` to 600 on all nodes.

  ```
  $ echo 600 > /sys/module/ptlrpc/at_max
  ```

- In modprobe.conf, run:

  ```
  options ptlrpc at_max=600
  ```

  The modprobe.conf line should be added (s/run/add) on all nodes before Lustre modules are loaded.

To disable adaptive timeouts, at run time, set `at_max` to 0 on all nodes.

```
$ echo 0 > /sys/module/ptlrpc/at_max
```

2. In Lustre 1.8, adaptive timeouts will be enabled, by default.

Note – Changing adaptive timeouts status at runtime may cause transient timeout, reconnect, recovery, etc.

22.1.3.2 Interpreting Adaptive Timeout Information

Adaptive timeout 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 /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:~# cat /proc/fs/lustre/osc/lustre-OST0001-osc-ce129800/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:~# cat /proc/fs/lustre/mdt/MDS/mds/stats
...
req_timeout              6 samples [sec] 1 10 15 105
...
```

22.1.4 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
```

22.1.5 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:

```
$ ctl 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.

22.1.5.1 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 OSTs' available blocks, speed, and the number of existing objects, etc. Using these criteria, the MDS chooses or avoids some OSTs for file object storage.

- **RR**

  Round-Robin (RR) allocates objects evenly across all OSTs. The round-robin stripe allocator is faster than QOS, and maximizes network balancing and improved 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

22.2 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).

22.2.1 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 performace, 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.

22.2.2 Watching the Client RPC Stream

In the same directory is a file that gives a histogram of the make-up of previous RPCs.

```
# 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
```

RPCs in flight

This represents the number of RPCs that are issued by the OSC but are not complete at the time of the snapshot. It should always be less than or equal to `max_rpcs_in_flight`.

pending {read,write} pages

These fields show the number of pages that have been queued for I/O in the OSC.

other RPCs in flight when a new RPC is sent

When an RPC is sent, it records the number of other RPCs that were pending in this table. When the first RPC is sent, the 0: row will be incremented. If the first RPC is sent while another is pending the 1: row will be incremented and so on. The number of RPCs that are pending as each RPC *completes* is not tabulated. This table is a good way of visualizing 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.

pages in each RPC

As an RPC is sent, the number of pages it is made of is recorded in order in this table. A single page RPC increments the 0: row, 128 pages the 7: row and so on.

These histograms can be cleared by writing any value into the `rpc_stats` file.

22.2.3 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

22.2.4 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
```

22.2.5 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.

22.2.6 Using File Readahead and Directory Statahead

Lustre 1.6.5.1 introduces 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.

22.2.6.1 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().

22.2.6.2 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.

22.2.7 mballoc History

/proc/fs/ldiskfs/sda/mb_history

mballoc stands for Multi-Block-Allocate. It is Lustre's ability to ask ext3 to allocate multiple blocks with a single request to the block allocator. Normally, an ext3 file system can allocate only one block per time. Each mballoc-enabled partition has this file. 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
2838  32779     4/12288/1   5/903/1     31       2   1  \           0   0

pid   inode     goal        result      found    grpscr \   merge   tailbroken
2838  32780     4/12288/1   5/904/1     31       2   1  \           1   8
2838  32781     4/12288/1   5/905/1     31       2   1  \           0   0
2838  32782     4/12288/1   5/906/1     31       2   1  \           1   2
2838  32783     4/12288/1   5/907/1     31       2   1  \           0   0
2838  32784     4/12288/1   5/908/1     31       2   1  \           1   4
2838  32785     4/12288/1   5/909/1     31       2   1  \           0   0
2838  32786     4/12288/1   5/910/1     31       2   1  \           1   2
2838  32787     4/12288/1   5/911/1     31       2   1  \           0   0
2838  32788     4/12288/1   5/912/1     31       2   1  \           1   16
2838  32789     4/12288/1   5/913/1     31       2   1  \           0   0
2828  32790     4/12288/1   5/914/1     31       2   0  \           1   2
2838  32791     4/12288/1   5/915/1     31       2   1  \           0   0
2838  32792     4/12288/1   5/916/1     31       2   1  \           1   4
2838  32793     4/12288/1   5/917/1     31       2   1  \           0   0
2838  32794     4/12288/1   5/918/1     31       2   1  \           1   2
2838  32795     4/12288/1   5/919/1     31       2   1  \           0   0
2828  32796     4/12288/1   5/920/1     31       2   1  \           1   8
2838  32797     4/12288/1   5/921/1     31       2   1  \           0   0
2838  32798     4/12288/1   5/922/1     31       2   1  \           1   2
2838  32799     4/12288/1   5/923/1     31       2   1  \           0   0
2838  32800     4/12288/1   5/924/1     31       2   1  \           1   4
```

```
pid     inode   goal        result      foundgrps   cr  \   merge   tailbroken
2838    32801   4/12288/1   5/925/1     31      2   1   \           0   0
2838    32802   4/12288/1   5/926/1     31      2   1   \           1   2
2838    32803   4/12288/1   5/927/1     31      2   1   \           0   0
2838    32804   4/12288/1   5/928/1     31      2   1   \           1   32
2838    32805   4/12288/1   5/929/1     31      2   1   \           0   0
2838    32806   4/12288/1   5/930/1     31      2   1   \           1   2
2838    32807   4/12288/1   5/931/1     31      2   1   \           0   0
2838    24579   3/12288/1   3/12289/1   11      1   1   \           0   0
```

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.
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.

22.2.8 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 currently 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.

22.2.9 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 maximum is 50 locks/MB. If there is too much memory pressure, then the LRU size is shrunk.

- 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.

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.

22.3 Debug Support

/proc/sys/lnet/debug

By default, Lustre generates a detailed log of all its 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[3] to improve performance. Raise the debug level when you need to collect the logs for debugging problems.

You can verify the debug level used by examining the sysctl that controls the debugging as shown below:

```
# sysctl lnet.debug
lnet.debug = -1
```

In the above example, -1 indicates full debugging; it is a bitmask. You can disable debugging completely by running the following command on all the concerned nodes:

```
# sysctl -w lnet.debug=0
lnet.debug = 0
```

The appropriate debug level for a production environment is 0x3f0400. It collects enough high-level information to aid debugging, but it does not cause any serious performance impact.

You can also verify and change the debug level using the `/proc` interface in Lustre as shown below:

```
# cat /proc/sys/lnet/debug
```

And change it to:

```
# echo 0x3f0400 > /proc/sys/lnet/debug
```

/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}`

3. 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.

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.

22.3.1 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
```

The OST .../stats files can be used to track the performance of RPCs that the OST gets from all clients. It is possible to get a periodic dump of values from these files, for instance every 10s, that show the RPC rates (similar to iostat) by using the llstat.pl tool like:

```
# 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
```

You can clear the stats by giving the -c option to llstat.pl. You can also mention how frequently (after how many seconds) it should clear the stats by mentioning an integer in -i option. For example, following is the output with -c and -i10 (stats for every 10 seconds):

```
$ 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)

22.3.1.1 llobdstat

The `llobdstat` utility parses obdfilter statistics files located at:

`/proc/fs/lustre/<ostname>/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
```

CHAPTER 23

Lustre Debugging

This chapter describes tips and information to debug Lustre, and includes the following sections:

- Lustre Debug Messages
- Tools for Lustre Debugging
- Troubleshooting with strace
- Looking at Disk Content
- Ptlrpc Request History

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`. The default patch after Lustre boots 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

Masks in /proc/sys/portals/subsystem_debug and /proc/sys/portals/debug controls the amount of information printed to the kernel debug logs. The subsystem_debug mask controls the subsystems (example: obdfilter, net, portals, OSC, etc.) and the debug mask controls the debug types written out to the log (example: info, error, trace, alloc, etc.).

To turn off Lustre debugging:

```
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 <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.
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.

Macro	Description
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".
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.

23.2.9 Debugging in UML

Lustre developers use gdb in User Mode Linux (UML) to debug Lustre. The `lmc` and `lconf` tools can be used to configure a Lustre cluster, load the required modules, start the services, and set up all the devices. lconf puts the debug symbols for the newly-loaded module into `/tmp/gdb-localhost.localdomain` on the host machine. These symbols can be loaded into gdb using the source command in gdb.

```
symbol-file
delete
symbol-file /usr/src/lum/linux
source /tmp/gdb-{hostname}
b panic
b stop
```

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, the debugfs tool can be used to obtain the attributes of all objects from different OST's. A sample run for the /mnt/lustre/frog file used in the example above 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**

Free Space and Quotas

This chapter describes free space and using quotas, and includes the following sections:

- Querying File System Space
- Using Quotas

24.1 Querying File System Space

The `lfs df` command is used to determine available disk space on a file system. It displays the amount of available disk space on the mounted Lustre file system and shows 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 `fs 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]
```

24.2 Using Quotas

The `lfs quota` command displays disk usage and quotas. By default, only user quotas are displayed (or with the `-u` flag).

A root user can use the `-u` flag, with the optional user parameter, to view the limits of other users. Users without root user authority can use the `-g` flag, with the optional group parameter, to view the limits of groups of which they are members.

Note – If a user has no files in a file system on which they have a quota, the `lfs quota` command shows `quota: none for the user`. The user's actual quota is displayed when the user has files in the file system.

Examples

To display quotas as user "bob," run:

```
$ lfs quota -u  /mnt/lustre
```

The above command displays disk usage and limits for user "bob."

To display quotas as root user for user "bob," run:

```
$ lfs quota -u bob /mnt/lustre
```

The system can also show the below information about disk usage by "bob."

To display your group's quota as "tom":

```
$ lfs -g tom /mnt/lustre
```

To display the group's quota of "tom":

```
$ lfs quota -g tom /mnt/lustre
```

Note – As for ext3, Lustre makes a sparse file in case you truncate at an offset past the end of the file. Space is utilized in the file system only when you actually write the data to these blocks.

CHAPTER **25**

Striping and I/O Options

This chapter describes file striping and I/O options, and includes the following sections:

25.1 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.

25.1.1 Advantages of Striping

There are two reasons to create files of multiple stripes: bandwidth and size.

25.1.1.1 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. In our experience, the requirement is "as quickly as possible," which usually means all OSSs.

Note – This assumes that your application is using enough client nodes, and can read/write data fast enough to take advantage of this much OSS bandwidth. The largest useful stripe count is bounded by the I/O rate of your clients/jobs divided by the performance per OSS.

25.1.1.2 Size

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 Free Space Management.

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.

25.1.2 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.

25.1.2.1 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.

25.1.2.2 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.

25.1.3 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. 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).

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.

25.2 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.

25.3 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

If you pass a stripe-size of 0, the file system's default stripe size is used. Otherwise, the stripe-size must be a multiple of 64 KB.

stripe-start

If you pass a starting-ost of -1, a random first OST is chosen. Otherwise, the file starts on the specified OST index, starting at zero (0).

stripe-count

If you pass a stripe-count of 0, the file system's default number of OSTs is used. A stripe-count of -1 means that all available OSTs should be used.

Note – If you pass a starting-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>
```

25.3.1 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).

25.3.2 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

25.3.3 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
```

25.4 Free Space Management

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.

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.)

25.4.1 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. 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	

25.4.2 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.

25.4.3 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.

25.5 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%).

25.5.1 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`

25.6 Other I/O Options

This section describes other I/O options, including end-to-end client checksums.

25.6.1 End-to-End Client Checksums

To guard against data corruption on the network, a Lustre client can perform end-to-end data checksums. This computes a 32-bit checksum of the data read or written on both the client and server, and ensures 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, 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 5 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 checksums on a client, run:

```
echo 1 > /proc/fs/lustre/llite/<fsname>/checksum_pages
```

To disable checksums on a client, run:

```
echo 0 > /proc/fs/lustre/llite/<fsname>/checksum_pages
```

To check the status of checksum, run:

```
lctl get_param osc.*.checksums
```

If it is set to 1, checksumming is enabled. If it is set to 0, checksumming is disabled.

25.6.1.1 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 checksum algorithm being used by Lustre, run:

```
$ echo <algorithm name> /proc/fs/lustre/osc/<fsname>-OST<index>- \
osc-*/checksum_type
```

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
```

25.7 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 **26**

Lustre Security

This chapter describes Lustre security and includes the following section:

- Using ACLs
- Using Root Squash

26.1 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.

26.1.1 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.

26.1.2 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.

26.1.3 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::---
```

26.2 Using Root Squash

Lustre 1.6 introduces root squash functionality, a security feature which controls super user access rights to an Lustre file system. 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.

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 Lusre cofiguration management server (MGS). The root squash feature also enables the Lustre administrator to specify a set of client for which UID/GID re-mapping does not apply.

26.2.1 Configuring Root Squash

Root squash functionality is managed by two configuration parameters, `root_squash` and `nosquash_nids`.

- The `root_squash` parameter specifies the UID and GID with which the root user accesses the Lustre file system.
- The `nosquash_nids` parameter specifies the set of clients to which root squash does not apply. LNET NID range syntax is used for this parameter (see the NID range syntax rules described in Enabling and Tuning Root Squash). For example:

  ```
  nosquash_nids=172.16.245.[0-255/2]@tcp
  ```

  In this example, root squash does not apply to TCP clients on subnet 172.16.245.0 that have an even number as the last component of their IP address.

26.2.2 Enabling and Tuning Root Squash

The default value for `nosquash_nids` 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.root_squash=500:501" \
           --param "mdt.nosquash_nids='0@elan1 192.168.1.[10,11]'" /dev/sda1
```

Root squash parameters can also be changed on an umounted device with `tunefs.lustre`. For example:

```
tunefs.lustre --param "mdt.root_squash=65534:65534"  \
--param "mdt.nosquash_nids=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.root_squash="1000:100"
lctl conf_param Lustre.mdt.nosquash_nids="*@tcp"
```

Note – When using the `lctl conf_param` command, keep in mind:

* `lctl conf_param` must be run on a live MGS
* `lctl conf_param` causes the parameter to change on all MDSs
* `lctl conf_param` is to be used once per a parameter

The `nosquash_nids` list can be cleared with:

```
lctl conf_param Lustre.mdt.nosquash_nids="NONE"
```

- OR -

```
lctl conf_param Lustre.mdt.nosquash_nids="clear"
```

If the `nosquash_nids` value consists of several NID ranges (e.g. 0@elan, 1@elan1), the list of NID ranges must be quoted with single (') or double ('') quotation marks. List elements must be separated with a space. For example:

```
mkfs.lustre ... --param "mdt.nosquash_nids='0@elan1 1@elan2'" /dev/sda1
lctl conf_param Lustre.mdt.nosquash_nids="24@elan 15@elan1"
```

These are examples of incorrect syntax:

```
mkfs.lustre ... --param "mdt.nosquash_nids=0@elan1 1@elan2" /dev/sda1
lctl conf_param Lustre.mdt.nosquash_nids=24@elan 15@elan1
```

To check root squash parameters, use the `lctl get_param` command:

```
lctl get_param mdt.Lustre-MDT0000.root_squash
lctl get_param mdt.Lustre-MDT000*.nosquash_nids
```

Note – An empty `nosquash_nids` list is reported as NONE.

26.2.3 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. That is, be careful doing root squash tuning.
- 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 `root_squash` parameter should be specified as `<decnum>':'<decnum>`. The `nosquash_nids` parameter should follow LNET NID range list syntax.

  LNET NID range syntax:

```
<nidlist>  :== <nidrange> [ ' ' <nidrange> ]
<nidrange> :== <addrrange> '@' <net>
<addrrange>:== '*' |
               <ipaddr_range> |
               <numaddr_range>
<ipaddr_range>:==
<numaddr_range>.<numaddr_range>.<numaddr_range>.<numaddr_range>
<numaddr_range>:== <number> |
                   <expr_list>
<expr_list>:== '[' <range_expr> [ ',' <range_expr>] ']'
<range_expr>:== <number> |
               <number> '-' <number> |
               <number> '-' <number> '/' <number>
<net>      :== <netname> | <netname><number>
<netname>  :== "lo" | "tcp" | "o2ib" | "cib" | "openib" | "iib" |
               "vib" | "ra" | "elan" | "gm" | "mx" | "ptl"
<number>   :== <nonnegative decimal> | <hexadecimal>
```

Note – For networks using numeric addresses (e.g. elan), the address range must be specified in the <numaddr_range> syntax. For networks using IP addresses, the address range must be in the <ipaddr_range>. For example, if elan is using numeric addresses, 1.2.3.4@elan is incorrect.

CHAPTER **27**

Lustre Operating Tips

This chapter describes tips to improve Lustre operations and includes the following sections:

27.1 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`

27.2 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
```

27.3 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

27.4 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.

27.5 Improving Lustre Metadata Performance While Using Large Directories

To improve metadata performance while using large directories, follow these tips:

- Have more 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 **28**

User Utilities (man1)

This chapter describes user utilities and includes the following sections:

28.1 lfs

The `lfs` utility can be used to create a new file with a specific striping pattern, determine the default striping pattern, 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 N] [[!] --group|-G <name>]
   [[!] --uid|-u N] [[!] --user|-U <name>]
   [[!] --pool <name>]
   <dirname|filename>
lfs getstripe [--obd|-O <uuid>] [--quiet|-q] [--verbose|-v]
              [--recursive|-r] <dirname|filename>
lfs setstripe [--size|-s stripe-size] [--count|-c stripe-count]
              [--offset|-o start-ost] [--pool|-p pool-name]
              <dirname|filename>
lfs setstripe -d <dirname>
lfs poollist <filename[.<pool>] | <pathname>
lfs quota [-v][-o obd_uuid] [-u|-g] <username|groupname> <filesystem>
lfs quota <filesystem>
lfs quota -t [-u|-g] <filesystem>
lfs quotacheck [-ug] <filesystem>
lfs quotachown [-i] <filesystem>
lfs quotaon [-ugf] <filesystem>
lfs quotaoff [-ug] <filesystem>
lfs quotainv [-ug] <filesystem>
lfs quotaver [-on] <filesystem>
lfs setquota  [-u|--user|-g|--group] <username|groupname>
              [--block-softlimit <block-softlimit>]
              [--block-hardlimit <block-hardlimit>]
              [--inode-softlimit <inode-softlimit>]
              [--inode-hardlimit <inode-hardlimit>]
              <filesystem>
```

```
lfs setquota [-u|--user|-g|--group] <username|groupname>
             [-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, `<filesystem>` refers to the mount point of the Lustre file system. The default is `/mnt/lustre`.

Note – The old `lfs quota` output was very detailed and contained clusterwide quota statistics (including clusterwide limits for a user/group and clusterwide usage for a user/group), as well as statistics for each MDS/OST. Now, `lfs quota` has been updated to provide only clusterwide statistics, by default. To obtain the full report of clusterwide limits, usage and statistics, use the `-v` option with `lfs quota`.

Description

The `lfs` utility is used to create a file with a specific pattern. It can be invoked interactively without any arguments or in a non-interactive mode with one of the arguments supported.

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 MDS or OSTs (as specified in the command) or all of the servers (MDS and OSTs).
df	
	Reports file system disk space usage or inode usage of each MDT/OST.
find	
	Searches the directory tree rooted at the given directory/filename for files that match the given parameters. 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.
--mtime	
	File status was last modified N*24 hours ago.
--ctime	
	File data was last changed N*24 hours ago.
--maxdepth	
	Allows the **find** command to descend at most N levels of the directory tree.
--print	
	Prints the full filename, followed by a new line.

Option	Description
--print0	
	Prints the full filename, followed by a null (\0) character.
--obd	
	File has an object on a specific OST(s).
--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]).
--gid	
	File has a specific group ID.
--group	
	File belongs to a specific group (numeric group ID allowed).
--uid	
	File has a specific numeric user ID.
--user	
	File owned by a specific user (numeric user ID allowed).
--pool	
	Specifies a pool to which a file must belong. Use --pool='' to find files that are not in any pool. Use --pool='*' to find files that are in any pool, excluding files that are not in a pool.
getstripe	
	Lists the striping information for a given filename or files in a directory (optionally recursive) for all files in a directory tree.
--quiet	
	Does not print object IDs.
--verbose	
	Prints striping parameters.
--recursive	
	Recurses into sub-directories.

Option	Description
setstripe	
	Create a new file or sets the directory default with the specified striping parameters.†
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, m or g (in KB, MB or GB, respectively).
stripe-count	
	Number of OSTs to stripe a file over. A stripe count of 0 uses the file system-wide default stripe count (1). A stripe count of -1 stripes over all available OSTs.
stripe-ost *	
	The OST index (base 10, starting at 0) on which to start striping for this file. A start-ost of -1 allows the MDS the choose the starting index. Selecting this value is strongly recommended, as this allows space and load balancing to be done by the MDS as needed.
pool-name	
	Name of the pre-defined pool of OSTs (see lctl) that will be used for striping. The **stripe-count**, **stripe-size** and **start-ost** values are used as well. The **start-ost** must be part of the pool or an error is returned.
setstripe -d	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] [-u|-g] <username|groupname> <filesystem>	
	Displays disk usage and limits, either for the full file system or for objexts on a specific OBD. A user or group name 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>	

Option	Description
	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).
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.
quotaver	
	Switches to new quota mode (-n) or old quota mode (-o).
setquota [-u | -g] <name> **[--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	

Option	Description
	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/file
```

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 will use the default striping pattern.

```
$ 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 setstripe --pool my_pool -c 2 /mnt/lustre/file
```

Creates a file striped on two OSTs from the pool `my_pool`

```
$ lfs pool_list /mnt/lustre/
```

Lists the pools defined for the mounted Lustre file system `/mnt/lustre`

```
$ lfs pool_list 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 sub-directories.

```
$ lfs find /mnt/lustre -mtime +30 -type f -print
```

Recursively lists all regular files in a given directory that are 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.

```
$ 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 of the OSTs.

```
$ 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 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.

28.2 lfsck

The e2fsprogs package contains an lfsck tool which does distributed coherency checking for the Lustre file system after e2fsck is run. In most cases, e2fsck is sufficient to repair any file system issues and lfsck is not required. To avoid lengthy downtime, you can also run lfsck once Lustre is already started.

Synopsis

```
lfsck [-h|--help] [-n|--nofix] [-l|--lostfound][-d|--delete] [-f|
--force] [-v|--verbose] --mdsdb mdsdb --ostdb ost1db [ost2db...]
<filesystem>
```

Note – As shown, <filesystem> refers to the Lustre file system mount point. The default is `/mnt/lustre`.

Options

The options and descriptions for the lfsck command are listed below.

Option	Description
-n	
	Performs a read-only check; does not repair the file system.
-l	
	Puts orphaned objects into a lost+found directory in the root of the file system.
-d	
	Deletes orphaned objects from the file system. Since objects on the OST are usually only one of several stripes of a file, it is often difficult to put multiple objects back together into a single usable file.
-h	
	Prints a brief help message.
--mdsdb ms_database_file	
	MDS database file created by running e2fsck --mdsdb mds_database_file device on the MDS backing device.
--ostdb ost1_database_file[,ost2_database_file,...]	
	OST database files created by running e2fsck --ostdb ost_database_file device on each OST backing device.

Description

If an MDS or an OST becomes corrupt, you can run a distributed check on the file system to determine what sort of problems exist.

1. **Run 'e2fsck -f' on the individual MDS / OST that had problems to fix any local file system damage.**

   It is a very good idea to run this e2fsck under "script" so you have a log of whatever changes it made to the file system (in case this is needed later). After this is complete, you can bring the file system up if necessary to reduce the outage window.

2. **Run a full e2fsck of the MDS to create a database for lfsck.**

   The `-n` option is critical for a mounted file system, otherwise you might corrupt your file system. 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 fastest if this is written to a local file system because of the seeking and small writes. Depending on the number of files, this step can take several hours to complete. In the following example, `/tmp/mdsdb` is the database file.

   ```
   e2fsck -n -v --mdsdb /tmp/mdsdb /dev/{mdsdev}
   ```

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
```

3. **Make this file accessible on all OSTs (either via a shared file system or by copying it to the OSTs – pdcp is very useful here. It copies files to groups of hosts and in parallel, it gets installed with pdsh. You can download it at:**

http://sourceforge.net/projects/pdsh)

Run a similar e2fsck step on the OSTs. You can run this step simultaneously on OSTs. The mdsdb is read-only in this step—a single copy can be shared by all OSTs.

```
e2fsck -n -v --mdsdb /tmp/mdsdb --ostdb /tmp/{ostNdb} /dev/{ostNdev}
```

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
```

Make the mdsdb and all of the ostdb files available on a mounted client so lfsck can be run to examine the file system and, optionally, correct defects that it finds.

```
lfsck -n -v --mdsdb /tmp/mdsdb --ostdb /tmp/{ost1db},{ost2db},... \
/lustre/mount/point
```

Example:

```
lfsck -n -v --mdsdb /home/mdsdb --ostdb /home/ostdb \
/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` does not repair any inconsistencies it finds, it only reports errors. It checks for three kinds of inconsistencies:

- Inode exists but has missing objects = dangling inode. Normally, this happens if there was a problem with an OST.
- Inode is missing but the OST has unreferenced objects = orphan object. Normally, this happens if there was a problem with the MDS.
- Multiple inodes reference the same objects. This happens if there was corruption on the MDS or if the MDS storage is cached and loses some, but not all, writes.

If the file system is busy, `lfsck` may report inconsistencies where none exist because of files and objects being created/removed after the database files were collected. Examined the results closely; you probably want to contact Lustre Support for guidance.

The easiest problem to resolve is orphaned objects. Use the `-l` option to `lfsck` so it links these objects to new files and puts them into `lost+found` in the Lustre file system, where they can be examined and saved or deleted as necessary. If you are certain that the objects are not necessary, `lfsck` can run with the `-d` option to delete orphaned objects and free up any space they are using.

To fix dangling inodes, `lfsck` creates new zero-length objects on the OSTs if the `-c` option is given. These files read back with binary zeros for the stripes that had objects recreated. Such files can also be read even without `lfsck` repair by using this command, run:

```
$ 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` recreate the objects. Generally, it is easier to delete these files and restore them from backup.

To fix inodes with duplicate objects, `lfsck` copies the duplicate object to a new object, and assign that to one of the files if the `-c` option is given. One of the files will be okay, and one will likely contain garbage; `lfsck` cannot, by itself, tell which one is correct.

28.3 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
```

28.4 Mount

Lustre uses the standard Linux `mount` command, and also supports a few extra options. For Lustre 1.4, the server-side options should be added to the XML configuration with the `–mountfsoptions= argument`.

Here are the Lustre-specific options:

Sever options	Description
extents	Use extent-mapped files
mballoc	Use Lustre file system allocator (required)

Lustre 1.6 server options	Description
abort_recov	Abort recovery when starting a target (currently an lconf option)
nosvc	Start only MGS/MGC servers
exclude	Start with a dead OST

Client options	Description
flock	Enable/disable flock support
user_xattr/nouser_xattr	Enable/disable user-extended attributes
retry=	Number of times a client will retry mount

28.5 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 **29**

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

29.1 User/Group Cache Upcall

This section describes user and group upcall.

Note – For information on a universal UID/GID, see Universal UID / GID.

29.1.1 Name

Use `/proc/fs/lustre/mds/mds-service/group_upcall` to look up a given user's group membership.

29.1.2 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.

29.1.2.1 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.

29.1.3 Parameters

- Name of the MDS service
- Numeric UID

29.1.4 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 30

Setting Lustre Properties (man3)

This chapter describes how to use `llapi` to set Lustre file properties.

30.1 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:

llapi_file_create

llapi_file_get_stripe

llapi_file_open

llapi_quotactl

30.1.1 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 4MB.

```
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;
}
```

30.1.2 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.

30.1.3 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.

30.1.4 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 clusterwide 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 mus 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 the 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.

30.1.5 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 **31**

Configuration Files and Module Parameters (man5)

This section describes configuration files and module parameters and includes the following sections:

- Introduction
- Module Options

31.1 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.)

31.2 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.

31.2.1 LNET Options

This section describes LNET options.

31.2.1.1 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  "
```

31.2.1.2 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.

31.2.1.3 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.

31.2.1.4 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.

31.2.2 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.

31.2.3 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.

31.2.4 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).

31.2.5 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.

31.2.6 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.

31.2.7 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.

31.2.8 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.
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).

31.2.9 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 **32**

System Configuration Utilities (man8)

This chapter includes system configuration utilities and includes the following sections:

32.1 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.

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.
--dryrun	Only prints what would be done; it does not affect the disk.

Option	Description
--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.
--reformat	
	Reformats an existing Lustre disk.

Option	Description
--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
```

32.2 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 1.6) 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.

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*	

Option	Description
	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.
--verbose	
	Prints more information.
tunefs.lustre --param="failover.node=192.168.0.13@tcp0" /dev/sda	
	Upgrades an old 1.4.x Lustre MDT to Lustre 1.6. The new file system name is "testfs".
tunefs.lustre --writeconf --mgs --mdt --fsname=testfs /dev/sda1	
	Upgrades an old 1.4.x Lustre MDT to Lustre 1.6, and starts with brand-new 1.6 configuration logs. All old servers and clients must be stopped.

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
```

Upgrading an old 1.4.x Lustre MDT to 1.6. The name of the new file system is `testfs`.

```
tunefs.lustre --mgs --mdt --fsname=testfs /dev/sda
```

Upgrading an old 1.4.x Lustre MDT to 1.6, and start with brand-new 1.6 configuration logs. All old servers and clients must be stopped.

```
tunefs.lustre --writeconf --mgs --mdt --fsname=testfs /dev/sda1
```

32.3 lctl

The lctl utility is used to 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.

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.
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.
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.

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.

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 dynamical 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
```

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
```

32.4 mount.lustre

The mount.lustre utility starts a Lustre client or target service.

Synopsis

```
mount -t lustre [-o options] device dir
```

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 is a client mount command used to mount the Lustre file system named <**fsname**> by contacting the Management Service at <**mgsspec**>. The format for <**mgsspec**> is defined below.
<disk_device>	This starts the target service defined by the mkfs.lustre command on the physical disk <disk_device>.

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.
noflock	Disables flock support.
user_xattr	Enables get/set user xattr.
nouser_xattr	Disables user xattr.
acl	Enables ACL support.
noacl	Disables ACL support.

In addition to the standard mount options and backing disk type (e.g. LDISKFS) options, Lustre understands the following server-specific options:

Option	Description
nosvc	Starts the MGC (and MGS, if co-located) for a target service, not the actual service.
mount -t lustre /dev/sda1 /mnt/test/mdt	Starts the Lustre target service on /dev/sda1.
mount -t lustre -L testfs-MDT0000 -o abort_recov /mnt/test/mdt	Starts the testfs-MDT0000 service (by using the disk label), but aborts the recovery process.

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 NID `cfs21@tcp0`.

```
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
```

32.5 New Utilities in Lustre 1.6

This section describes new utilities available in Lustre 1.6.

32.5.1 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.

32.5.2 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.

32.5.3 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 Lustre.

32.5.4 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.

32.5.5 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 also includes 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
  ```

32.5.6 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.

32.5.7 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.

32.5.7.1 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
```

32.5.8 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
```

32.5.9 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
```

32.5.10 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
```

32.5.11 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
```

32.5.12 plot-llstat

The plot-llstat utility plots Lustre statistics.

Synopsis

```
plot-llstat results_filename [parameter_index]
```

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.

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

Example

```
llstat -i2 -g -c lustre-OST0000 > log
plot-llstat log 3
```

32.5.13 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
```

32.5.14 ll_recover_lost_found_objs

The `ll_recover_lost_found_objs` utility helps recover Lustre OST objects from a lost and found directory.

Synopsis

```
$ ll_recover_lost_found_objs [-hv] -d directory
```

Description

The `ll_recover_lost_found_objs` utility recovers objects from a lost and found directory that might be created if an OST has a corrupted directory. 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. Using `ll_recover_lost_found_objs` enables you to recover these objects.

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 **33**

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:

33.1 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.

33.2 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.)

33.3 Minimum Stripe Size

Due to the 64 KB PAGE_SIZE on some 64-bit machines, the minimum stripe size is set to 64 KB.

33.4 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.

33.5 Maximum Number of Clients

Currently, the number of clients is limited to 131072. We have tested up to 22000 clients.

33.6 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 8 TB, regardless of whether 32-bit or 64-bit kernels are on the server. (For 2.6 kernels, the 8 TB limit is imposed by ext3). Currently, testing is underway to allow file systems up to 16 TB.

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.

33.7 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.

33.8 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.

33.9 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 to Format MDT and OST File Systems.

33.10 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.

33.11 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.

33.12 OSS RAM Size for a Single OST

For a single OST, there is no strict rule to size the OSS RAM. However, as a guideline, 1GB per OST is a reasonable RAM size. This provides sufficient RAM for the OS, and an appropriate amount (600 MB) for the metadata cache, which is very important for efficient object creation/lookup when there are many objects.

The minimum, recommended RAM size is 600 MB per OST, plus 500 MB for the metadata cache. In a failover scenario, you should double these sizes (therefore 1.2 GB per OST).

In this case, you have about 1.2GB/OST. It might be difficult to work with 1GB/primary OST as it gives 800MB/2OST which leaves only 100MB for a working set for each OST. This ends up as a maximum of ~ 2.4 million objects on the OST before it starts getting thrashed.

APPENDIX **A**

Version Log

Manual Version	Date	Details of Edits	Bug
1.16	04/20/09	1. Section 29.1.2.1 incorrect - default upcall changed	17571
		2. Lustre-1.6_man_v1.13, Section 19.2.2 (obdfilter_survey) is out of date	16697
		3. Lockless I/O tunables	17984
		4. Service Tags additions to the manual.	16032
		5. lst stat command syntax needs more details	17989
		6. Replace “star” references in Lustre documentation with “GNU tar”.	18354
		7. Update 'Removing on OST' procedure in the Lustre manual.	18263
		8. Request information concerning health check values.	18110
		9. Monitoring tools.	18242
		10. Document file readahead and directory statahead.	18542
		11. Document the root squash feature.	16519
		12. Some errors in “32.5.13 routerstat” of the Lustre Operations Manual	18712
		13. Kernel-ib must be installed on patchless clients	19300
		14. 22.1.5 Free Space Distribution of Lustre manual needs updating and clarification.	18543
		15. LNET routes statements of any significant size cause errors.	18766
1.15	11/21/08	1. Section 3.3.3 can be extended.	17268
		2. Lustre-1.6_man_v1.13 Section 19.2.2 is out of date.	16697

Manual Version	Date	Details of Edits	Bug
		3. /proc/sys/lnet/upcall - threat or menace?	16629
		4. Section 29.1.2.1 incorrect - default upcall changed.	17571
1.14	09/19/08	1. Update example "routes" parameters in Sec. 5.2.2.	16269
		2. URLs for Lustre kernel downloads are unwieldy.	15850
		3. mount-by-disklabel: add warning not to use in multipath environment.	16370
		4. Document file system incompatibility when using ldiskfs2.	12479
		5. Manual update may be needed for change in maximum number of clients.	16484
		6. lfs syntax updates in documentation.	16485
		7. Errors in 1.6 manual in 10.3 Creating an External Journal.	16543
		8. Re-word MGS failover note in 8.7.3.3 Failback.	16552
		9. Add statistics to monitor quota activity.	15058
		10. Documentation for filefrag using FIEMAP.	16708
		11. Document man pages for llobdstat(8), llstat(8), plot-llstat(8), l_getgroups(8), lst(8) and routerstat(8).	16725
		12. Update Lustre manual re: lru_size parameter.	16843
		13. LBUG information missing.	16820
		14. Re-write LNET self-test topic.	16567
		15. Update Lustre manual for "lctl{set,get}_param"	15171
1.13	07/03/08	1. fsname maximum length not documented.	15486
		2. Granted cache affects accuracy of lquota, record it in the manual.	15438
		3. Replace 'striping pattern' instances with 'file layout'	15755
		4. Update manual content re: forced umount of OST in failover case.	15854
		5. Verify URL for PIOS in manual section 19.3 PIOS test tool.	15955
		6. Adaptive timeout documentation corrections.	16039
		7. mkfs.lustre man page may contain a small error	15832
		8. Missing parameters in lctl document.	13477
		9. Merge Lustre debugging information.	12046

Manual Version	Date	Details of Edits	Bug
		10. Need to add Lustre mount parameters to manual.	14514
		11. Multi-rail LNET configuration.	14534
		12. Lustre protocols and Wireshark	12161
		13. Loading lnet_selftest modules.	16233
		14. DRBD + Lustre performance measurements	14701
		15. Many documentation errors.	13554
1.12	04/21/08	1. Additional Lustre manual content - /proc entries.	15039
		2. Additional Lustre manual content - atime.	15042
		3. Additional Lustre manual content - building kernels.	15047
		4. Additional Lustre manual content - Lustre clients.	15048
		5. Additional Lustre manual content - compilation.	15050
		6. Additional Lustre manual content - Lustre configuration.	15051
		7. Additional Lustre manual content - Lustre debugging.	15053
		8. Additional Lustre manual content - e2fsck.	15054
		9. Additional Lustre manual content - failover.	15074
		10. Additional Lustre manual content - evictions.	15071
		11. Additional Lustre manual content - file systems.	15079
		12. Additional Lustre manual content - hardware.	15080
		13. Additional Lustre manual content - kernels.	15085
		14. Additional Lustre manual content - network issues.	15102
		15. Additional Lustre manual content - Lustre performance.	15108
		16. Additional Lustre manual content - quotas.	15110
		17. Update the Lustre manual for heartbeat content.	15158
		18. ksocklnd module parameter enable_irq_affinity now defaults to zero	15174
		19. Multiple mentions of /etc/init.d/lustre in manual.	15510
		20. Incorrect flag for tune2fs	15522
1.11	3/11/08	1. Updated content in Failover chapter.	12143

Manual Version	Date	Details of Edits	Bug
		2. Man pages for llapi_ functions.	12043
		3. DDN updates to the manual.	12173
		4. DDN configuration.	12142
		5. Update Lustre manual according to changes in BZ 12786.	13475
		6. Add lockless I/O tunables content to the Lustre manual.	13833
		7. Small error in LNET self-test documentation sample script.	14680
		8. LNET self-test	10916
		9. Documentation for Lustre checksumming feature.	12399
		10. Ltest OSTs seeing out-of-memory condition.	11176
		11. Section 7.1.3 Quota Allocation	14372
		12. localflock not documented.	13141
		13. Lustre group file quota does not error, allows files up to the hard limit.	13459
		14. Changing the quota of a user doesn't work.	14513
		15. Documentation errors.	13554
		16. Need details about old clients and new file systems.	14696
		17. Missing build instructions.	14913
		18. Update ip2nets section in Lustre manual and add example shown	12382
		19. Free space management	12175
1.10	12/18/07	1. Updated content in Disk Performance Measurement section of the RAID chapter.	12140
		2. Added lfs option to User Utilities chapter.	14024/ 12186
		3. Added supplementary group upcall content to the Lustre Programming Interfaces chapter	12680
		4. Added content (new section, Network Tuning) to the Lustre Tuning chapter.	10077
		5. Added new chapter, Lustre Debugging, to the Lustre manual	12046/ 13618

Manual Version	Date	Details of Edits	Bug
		6. Updated unlink and munlink command information in the Identifying a Missing OST topic in the Lustre Troubleshooting and Tips chapter.	14239
		7. Minor error in manual Chapter III - 3.2.3.3	14414
1.9	11/2/07	1. Updated content in the Bonding chapter.	n/a
		2. Updated content in the Lustre Troubleshooting and Tips chapter.	n/a
		3. Updated content in the Lustre Security chapter.	n/a
		4. Added PIOS Test Tool topic to the Lustre I/O Kit chapter.	11810
		5. Updated content in Chapter IV - 2. Striping and Other I/O Options, Striping Using ioctl section.	12032
		6. Updated content in Chapter III - 2. LustreProc, Section 2.2.3 Client Read-Write Offset Survey and Section 2.2.4 Client Read-Write Extents Survey.	12033
		7. Updated content in Chapter V - 4. System Configuration Utilities (man8), Section 4.3.4 Network commands.	12034
		8. Updated content in the Lustre Installation chapter.	12035
		9. Updated content in Chapter V - 1. User Utilities (man1), Section 1.2 fsck.	12036
		10. Updated content in RAID chapter.	12040/ 12070
		11. Updated content in Striping and Other I/O Options, lfs setstripe - Setting Striping Patterns section.	12042
		12. Updated content in Configuring the Lustre Network chapter.	12426
		13. Updated content in the System Limits chapter.	12492
		14. Updated content in the User Utilities (man1 chapter.	12799
		15. Updated content in the Lustre Configuration chapter.	13529
		16. Updated content in Section 4.1.11 of the Lustre Troubleshooting and Tips chapte.r	13810/ 11325/ 12164
		17. Updated content in Prerequisites and Lustre Installation chapters.	13851

Manual Version	Date	Details of Edits	Bug
		18. Updated content in the Starting LNET section, Configuring the Lustre Network chapter.	14024
1.8	09/29/07	1. Added new chapter (POSIX) to manual.	12048
		2. Added new chapter (Benchmarking) to manual.	12026
		3. Added new chapter (Lustre Recovery) to manual.	12049 / 12141
		4. Updated content in the Configuring Quotas chapter.	13433
		5. Updated content in the More Complicated Configurations chapter.	12169
		6. Updated content in the LustreProc chapter.	12385 / 12383 / 12039
		7. Corrected errors in Section 4.1.1.2.	12981
		8. Merged MXLND information from Myricom.	12158
		9. Updated content in the Configuring Lustre Examples chapter.	12136
		10. Updated content in the RAID chapter.	12170 / 12140
		11. Updated content in the Configuration Files Module Parameters chapter.	12299
1.7	08/30/07	1. Added mballoc3 content to the LustreProc chapter.	12384 / 10816
1.6	08/23/07	1. Updated content in the Expanding the file system by Adding OSTs section.	13118
		2. Updated content in the Failover chapter.	13022 / 12168 / 12143
		3. Added Mechanics of Lustre Readahead content.	13022
		4. Updated content in the Lustre Troubleshooting and Tips chapter.	12164 / 12037 / 12047 / 12045
		5. Updated content in the Free Space and Quotas chapter.	12037
		6. Updated content in the Lustre Operating Tips chapter.	12037
		7. Added a new appendix - Knowledge Base chapter.	12037

Manual Version	Date	Details of Edits	Bug
1.5	07/20/07	1. Updated content in the Lustre Installation chapter.	12037
		2. Updated content in the Failover chapter.	12037
		3. Updated content in the Bonding chapter.	12037
		4. Updated content in the Striping and I/O Options chapter.	12037/ 12025
		5. Updated content in the Lustre Operating Tips chapter.	12037
		6. Developmental edit of remaining chapters in semiannual.	11417
		7. Added new chapter (Lustre SNMP Module) to the manual.	12037
		8. Added new chapter (Backup and Recovery) to the manual.	12037
1.4	07/08/07	1. Added content to the Configuring Lustre Network chapter.	12037
		2. Added content to the LustreProc chapter.	12037
		3. Added content to the Lustre Troubleshooting and Tips chapter.	12037
		4. Added content to the Lustre Tuning chapter.	12037
		5. Added content to the Prerequisites chapter.	12037
		6. Completed re-development of index in manual.	11417
		7. Developmental edit of select chapters in manual.	11417
1.3	06/08/07	1. Updated section 2.2.1.1.	12483
		2. Added enhancements to the DDN Tuning chapter.	12173
		3. Updated the User Utilities (man1) chapter.	n/a
		4. Added lfsck and e2fsck content to the Lustre Programming Interfaces (man2) chapter.	12036
		5. Removed MDS Space Utilization content.	12483
		6. Added training slide updates to the manual.	12478
		7. Added enhancements to 8.1.5 Formatting section.	n/a
1.2	05/25/07	1. Added striping Using ioctl (Part IV, Chapter 2)	12032
		2. Added Client Read/Write Offset and Extents content (Part III, Chapter 2)	12033
		3. Added Building RPMs content (Part II, Chapter 2)	12035

Manual Version	Date	Details of Edits	Bug
		4. Added Setting the Striping Pattern content and I/O (Part IV, Chapter 2 - lfs setstripe)	12036
		5. Added Free Space Management content (Part III, Chapter 2 - 2.1.1/proc entries)	12175/ 12039/ 12028
		6. Added /proc content and I/O (Part III, Chapter 2 - 2.1.1 /proc entries)	12172
1.1	02/03/07	1. Upgraded all chapters Lustre 1.4 to 1.6.	
		2. Introduction and information of new features of Lustre 1.6 like MountConf, MGS, MGC, and so on.	
		3. Introduction and information of mkfs.lustre, mount.lustre, and tunefs.lustre utilities.	
		4. Removed lmc and lconf utilities.	
		5. Added Chapter II - 10. Upgrading Lustre from 1.4 to 1.6.	
		6. Removed Appendix Upgrading 1.4.5 to 1.4.6.	
		7. Added content on permanently removing an OST.	

APPENDIX **B**

Lustre Knowledge Base

The Knowledge Base is a collection of tips and general information regarding Lustre.

Can I run Lustre in a heterogeneous environment (32-and 64-bit machines)?

How to build and configure Infiniband support for Lustre

Can the same Lustre file system be mounted at multiple mount points on the same client system?

How do I identify files affected by a missing OST?

How-To: New Lustre network configuration

How to fix bad LAST_ID on an OST

Why can't I run an OST and a client on the same machine?

Information on the Socket LND (socklnd) protocol

Information on the Lustre Networking (LNET) protocol

Explanation of: '... previously skipped # similar messages' in Lustre logs

What should I do if I suspect device corruption (Example: disk errors)

How do I clean up a device with lctl?

What is the default block size for Lustre?

How do I determine which Lustre server (MDS/OST) was connected to a particular storage device?

Does the mount option "--bind" allow mounting a Lustre file system to multiple directories on the same client system?

What operations take place in Lustre when a new file is created?

Questions about using Lustre quotas

When mounting an MDT filesystm, the kernel crashes. What do I do?

How do I determine which Ethernet interfaces Lustre uses?

How can I check if a file system is active (the MGS, MDT and OSTs are all online)?

You can look at /proc/fs/lustre/lov/*/target_obds for "ACTIVE" vs "INACTIVE" on MDS/clients.

How to reclaim the 5 percent of disk space reserved for root?

If your file system normally looks like this:

```
$ df -h /mnt/lustre
Filesystem          Size   Used   Avail  Use%   Mounted on
databarn            100G   81G    14G    81%    /mnt/lustre
```

You might be wondering: where did the other 5 percent go? This space is reserved for the root user.

Currently, all Lustre installations run the ext3 file system internally on service nodes. By default, ext3 reserves 5 percent of the disk for the root user.

To reclaim this space for use by all users, run this command on your OSSs:

```
tune2fs [-m reserved_blocks_percent] [device]
```

This command takes effect immediately. You do not need to shut down Lustre beforehand or restart Lustre afterwards.

Why are applications hanging?

The most common cause of hung applications is a timeout. For a timeout involving an MDS or failover OST, applications attempting to access the disconnected resource wait until the connection is re-established.

In most cases, applications can be interrupted after a timeout with the KILL, INT, TERM, QUIT, or ALRM signals. In some cases, for a command which communicates with multiple services in a single system call, you may have to wait for multiple timeouts.

How do I abort recovery? Why would I want to?

If an MDS or OST is not gracefully shut down, for example a crash or power outage occurs, the next time the service starts it is in "recovery" mode.

This provides a window for any existing clients to re-connect and re-establish any state which may have been lost in the interruption. By doing so, the Lustre software can completely hide failure from user applications.

The recovery window ends when either:

- All clients which were present before the crash have reconnected; or
- A recovery timeout expires

This timeout must be long enough to for all clients to detect that the node failed and reconnect. If the window is too short, some critical state may be lost, and any in-progress applications receive an error. To avoid this, the recovery window of Lustre 1.x is conservatively long.

If a client which was not present before the failure attempts to connect, it receives an error, and a message about recovery displays on the console of the client and the server. New clients may only connect after the recovery window ends.

If the administrator knows that recovery will not succeed, because the entire cluster was rebooted or because there was an unsupported failure of multiple nodes simultaneously, then the administrator can abort recovery.

With Lustre 1.4.2 and later, you can abort recovery when starting a service by adding --abort-recovery to the lconf command line. For earlier Lustre versions, or if the service has already started, follow these steps:

1. **Find the correct device. The server console displays a message similar to:**

   "RECOVERY: service mds1, 10 recoverable clients, last_transno 1664606"

2. **Obtain a list of all Lustre devices. On the MDS or OST, run:**

   ```
   lctl device_list
   ```

3. **Look for the name of the recovering service, in this case "mds1":**

   3 UP mds mds1 mds1_UUID 2

4. **Instruct Lustre to abort recovery, run:**

   ```
   lctl --device <OST device number> abort_recovery
   ```

   The device number is on the left.

What does "denying connection for new client" mean?

When service nodes are performing recovery after a failure, only clients which were connected before the failure are allowed to connect. This enables the cluster to first re-establish its pre-failure state, before normal operation continues and new clients are allowed to connect.

How do I set a default debug level for clients?

If using zeroconf (mount -t lustre), you can add a line similar to the following to your modules.conf:

```
post-install portals sysctl -w lnet.debug=0x3f0400
```

This sets the debug level, whenever the portals module is loaded, to whatever value you specify. The value specified above is a good starting choice, and will become the in-code default in Lustre 1.0.2, as it provides useful information for diagnosing problems without materially impairing the performance of Lustre.)

How can I improve Lustre metadata performance when using large directories (> 0.5 million files)?

On the MDS, more memory translates into bigger caches and, therefore, higher performance. One of the requirements for higher metadata performance is to have lots of RAM on the MDS.

The other requirement (if not running a 64-bit kernel) is to patch the core kernel on the MDS with the

3G/1G patch to increase the available kernel address space. This, again, translates into having support for bigger caches on the MDS.

Usually the address space is split in a 3:1 ratio (3G for userspace and 1G for kernel). The 3G/1G patch changes this ratio to 3G for kernel/1G for user (3:1) or 2G for kernel and 2G for user (2:2).

File system refuses to mount because of UUID mismatch

When Lustre exports a device for the first time on a target (MDS or OST), it writes a randomly-generated unique identifier (UUID) to the disk from the .xml configuration file. On subsequent exports of that device, the Lustre code verifies that the UUID on disk matches the UUID in the .xml configuration file.

This is a safety feature which avoids many potential configuration errors, such as devices being renamed after the addition of new disks or controller cards to the system, cabling errors, etc. This results in messages, such as the following, appearing on the system console, which normally indicates a system configuration error:

```
af0ac_mds_scratch_2b27fc413e does not match last_rcvd UUID
8a9c5_mds_scratch_8d2422aa88
```

In some cases, it is possible to get the incorrect UUID in the configuration file, for example by regenerating the .xml configuration file a second time. In this case, you must specify the device UUIDs when the configuration file is built with the --ostuuid or --mdsuuid options to match the original UUIDs instead of generating new ones each time.

```
lmc -add ost --node ostnode --lov lov1 --dev /dev/sdc --ostuuid
3dbf8_OST_ostnode_ddd780786b

lmc -add mds --node mdsnode --mds mds_scratch --dev /dev/sdc --mdsuuid
8a9c5_mds_scratch_8d2422aa88
```

How do I set up multiple Lustre file systems on the same node?

Assuming you want to have separate file systems with different mount locations, you need a dedicated MDS partition and Logical Object Volume (LOV) for each file system. Each LOV requires a dedicated OST(s).

For example, if you have an MDS server node, mds_server, and want to have mount points /mnt/foo and /mnt/bar, the following lines are an example of the setup (leaving out the --add net lines):

Two MDS servers using distinct disks:

```
lmc -m test.xml --add mds --node mds_server --mds foo-mds --group \
foo-mds --fstype ldiskfs --dev /dev/sda
lmc -m test.xml --add mds --node mds_server --mds bar-mds --group \
bar-mds --fstype ldiskfs --dev /dev/sdb
```

Now for the LOVs:

```
lmc -m test.xml --add lov --lov foo-lov --mds foo-mds \
--stripe_sz 1048576 --stripe_cnt 1 --stripe_pattern 0
lmc -m test.xml --add lov --lov bar-lov --mds bar-mds \
--stripe_sz 1048576 --stripe_cnt 1 --stripe_pattern 0
```

Each LOV needs at least one OST:

```
lmc -m test.xml --add ost --node ost_server --lov foo-lov \
--ost foo-ost1 --group foo-ost1 --fstype ldiskfs --dev /dev/sdc
lmc -m test.xml --add ost --node ost_server --lov bar-lov \
--ost bar-ost1 --group
bar-ost1 --fstype ldiskfs --dev /dev/sdd
```

Set up the client mount points:

```
lmc -m test.xml --add mtpt --node foo-client --path /mnt/foo \
--mds foo-mds --lov foo-lov
lmc -m test.xml --add mtpt --node bar-client --path /mnt/bar \
--mds bar-mds --lov bar-lov
```

If the Lustre file system "foo" already exists, and you want to add the file system "bar" without reformatting foo, use the group designator to reformat only the new disks:

```
ost_server> lconf --group bar-ost1 --select bar-ost1 \
--reformat test.xml
mds_server> lconf --group bar-mds --select bar-mds \
--reformat test.xml
```

If you change the --dev that foo-mds uses, you also need to commit that new configuration (foo-mds must not be running):

```
mds_server> lconf --group foo-mds --select foo-mds --write_conf
test.xml
```

Note – If you want both mount points on a client, you can use the same client node name for both mount points.

Is it possible to change the IP address of a OST? MDS? Change the UUID?

The IP address of any node can be changed, as long as the rest of the machines in the cluster are updated to reflect the new location. Even if you used hostnames in the xml config file, you need to regenerate the configuration logs on your metadata server.

It is also possible to change the UUID, but unfortunately it is not very easy as two binary files would need editing.

How do I set striping on a file?

To stripe a file across <n> OSTs with stripesize of <b> blocks per stripe, run:

```
lfs setstripe <new_filename> <stripe_size> <stripe_offset>
<stripe_count>
```

This creates "new_filename" (which must not already exist).

We strongly recommend that the stripe_size value be 1MB or larger (size in bytes). Best performance is seen with one or two stripes per file unless it is a file that has shared IO from a large number of clients, when the maximum number of stripes is best (pass -1 as the stripe count to get maximum striping).

The stripe_offset (OST index which holds the first stripe, subsequent stripes are created on sequential stripes) should be "-1" which means allocate stripes in a round-robin manner. Abusing the stripe_offset value leads to uneven usage of the OSTs and premature file system usage.

Most users want to use:

```
lfs setstripe <new_filename> 2097152 -1 N
```

Or use system-wide default stripe size:

```
lfs setstripe <new_filename> 0 -1 N
```

You may want to make a simple wrapper script that only accepts the <stripe_count> parameter. Usage info via "lfs help setstripe".

How do I set striping for a large number of files at one time?

You can set a default striping on a directory, and then any regular files created within that directory inherit the default striping configuration. To do this, first create a directory if necessary and then set the default striping in the same manner as you do for a regular file:

```
lfs setstripe <directory> <stripe_size> -1 <stripe_count>
```

If the stripe_size value is zero (0), it uses the system-wide stripe size. If the stripe_count value is zero (0), it uses the default stripe count. If the stripe_count value is -1, it stripes across all available OSTs. The best performance for many clients writing to individual files is at 1 or 2 stripes per file, and maximum stripes for large shared-I/O files (i.e. many clients reading or writing the same file at one time).

If I set the striping of N and B for a directory, do files in that directory inherit the striping or revert to the default?

All new files get the new striping parameters, and existing files will keep their current striping (even if overwritten). To "undo" the default striping on a directory (to use system-wide defaults again) set the striping to "0 -1 0".

Can I change the striping of a file or directory after it is created?

You cannot change the striping of a file after it is created. If this is important (e.g., performance of reads on some widely-shared large input file) you need to create a new file with the desired striping and copy the data into the old file. It is possible to change the default striping on a directory at any time, although you must have write permission on this directory to change the striping parameters.

How do I replace an OST or MDS?

The OST file system is simply a normal ext3 file system, so you can use any number of methods to copy the contents to the new OST.

If possible, connect both the old OST disk and new OST disk to a single machine, mount them, and then use rsync to copy all of the 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
```

The same can be done for the MDS, but it needs an 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
```

How do I configure recoverable / failover object servers?

There are two object server modes: the default failover (recoverable) mode, and the fail-out mode. In fail-out mode, if a client becomes disconnected from an object server because of a server or network failure, applications which try to use that object server will receive immediate errors.

In failover mode, applications attempting to use that resource pause until the connection is restored, which is what most people want. This is the default mode in Lustre 1.4.3 and later.

To disable failover mode:

1. **If this is an existing Lustre configuration, shut down all client, MDS, and OSS nodes.**
2. **Change the configuration script to add --failover to all "ost" lines.**

   Change lines like:

   ```
   lmc --add ost ...
   ```

   to:

   ```
   lmc --add ost ... --failover
   ```

   and regenerate your Lustre configuration file.
3. **Start your object servers.**

   They should report that recovery is enabled to syslog:

   ```
   Lustre: 1394:0:(filter.c:1205:filter_common_setup()) \
   databarn-ost3: recovery enabled
   ```
4. **Update the MDS and client configuration logs. On the MDS, run**:

   ```
   lconf --write_conf /path/to/lustre.xml
   ```
5. **Start the MDS as usual.**
6. **Mount Lustre on the clients.**

How do I resize an MDS / OST file system?

This is a method to back up the MDS, including the extended attributes containing the striping data. If something goes wrong, you can restore it to a newly-formatted larger file system, without having to back up and restore all OSS data.

Caution – If this data is very important to you, we strongly recommend that you try to back it up before you proceed.

It is possible to run out of space or inodes in both the MDS and OST file systems. If these file systems reside on some sort of virtual storage device (e.g., LVM Logical Volume, RAID, etc.) it may be possible to increase the storage device size (this is device-specific) and then grow the file system to use this increased space.

1. **Prior to doing any sort of low-level changes like this, back up the file system and/or device. See How do I backup / restore a Lustre file system?**
2. **After the file system or device has been backed up, increase the size of the storage device as necessary. For LVM this would be:**

   ```
   lvextend -L {new size} /dev/{vgname}/{lvname}
   ```

   or

   ```
   lvextend -L +{size increase} /dev/{vgname}/{lvname}
   ```

3. **Run a full e2fsck on the file system, using the Lustre e2fsprogs (available at the Lustre download site or http://downloads.clusterfs.com/public/tools/e2fsprogs/. Run:**

   ```
   e2fsck -f {dev}
   ```

4. **Resize the file system to use the increased size of the device. Run:**

   ```
   resize2fs -p {dev}
   ```

How do I backup / restore a Lustre file system?

Several types of Lustre backups are available.

CLIENT FILE SYSTEM-LEVEL BACKUPS

It is possible to back up Lustre file systems from a client (or many clients in parallel working in different directories), via any number of user-level backup tools like tar, cpio, Amanda, and many enterprise-level backup tools. However, due to the very large size of most Lustre file systems, full backups are not always possible. Doing backups of subsets of the file system (subdirectories, per user, incremental by date, etc.) using normal file backup tools is still recommended, as this is the easiest method from which to restore data.

TARGET RAW DEVICE-LEVEL BACKUPS

In some cases, it is desirable to do full device-level backups of an individual MDS or OST storage device for various reasons (before hardware replacement, maintenance or such). 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.

If hardware replacement is the reason for the backup or if there is a spare storage device then it is possible to just do a raw copy of the MDS/OST from one block device to the other as long as the new device is at least as large as the original device using the command:

```
dd if=/dev/{original} of=/dev/{new} bs=1M
```

If hardware errors are causing read problems on the original device then using the command below allows 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
```

Even in the face of hardware errors, the ext3 file system is very robust and it may be possible to recover file system data after e2fsck is run on the new device.

TARGET FILE SYSTEM-LEVEL BACKUPS

In other cases, it is desirable to make a backup of just the file data in an MDS or OST file system instead of backing up 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 file system need to be changed, to use less space for the backup, etc).

In this case it is possible to mount the ext3 file system directly from the storage device and do a file-level backup. Lustre MUST BE STOPPED ON THAT NODE.

To back up such a file system properly also requires that any extended attributes (EAs) stored in the file system be backed up, but unfortunately current backup tools do not properly save this data so an extra step is required.

1. **Make a mountpoint for the mkdir /mnt/mds file system.**
2. **Mount the file system there.**
   - For 2.4 kernels, run:

     ```
     mount -t ext3 {dev} /mnt/mds
     ```

   - For 2.6 kernels, run:

     ```
     mount -t ldiskfs {dev} /mnt/mds
     ```

3. **Change to the mount point being backed up. Type:**

   ```
   cd /mnt/mds
   ```

4. **Back up the EAs. Type:**

   ```
   getfattr -R -d -m '.*' -P . > ea.bak
   ```

   The getfattr command is part of the "attr" package in most distributions.

   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 contact us for assistance.

5. **Verify that the ea.bak file has properly backed up your EA data on the MDS.**

Without this EA data your backup is not useful. You can look at this file with "more" or a text editor, and it should have an item for each file like:

```
# file: ROOT/mds_md5sum3.txt
trusted.lov0s0AvRCwEAAABXoKUCAAAAAAAAAAAAAAAAAAAQAAEAAADD5QoAAAAAA
AAAAAAAAAAAAAAAAAAEAAAA=
```

6. **Back up all file system data. Type:**

   ```
   tar czvf {backup file}.tgz
   ```

7. **Change out of the mounted file system. Type.**

   ```
   cd -
   ```

8. **Unmount the file system. Type:**

   ```
   umount /mnt/mds
   ```

Follow the same process on each of the OST device file systems. The backup of the EAs (described in Step 4), is not currently required for OST devices, but this may change in the future.

To restore the file-level backup you need to format the device, restore the file data, and then restore the EA data.

9. **Format the new device. The easiest way to get the optimal ext3 parameters is to use lconf --reformat {config}.xml ONLY ON THE NODE being restored.**

   If there are multiple services on the node, then this reformats all of the devices on that node and should NOT be used. Instead, use the step below:

   - For MDS file systems, use: mke2fs -j -J size=400 -I {inode_size} -i 4096 {dev} where {inode_size} is at least 512, and possibly larger if you have a default, stripe count > 10 (inode_size = power_of_2_>=_than(384 + stripe_count * 24)).
   - For OST file systems, use: mke2fs -j -J size=400 -I 256 -i 16384 {dev}

10. **Enable ext3 file system directory indexing. Type:**

    ```
    tune2fs -O dir_index {dev}
    ```

11. **Mount the file system. Type:**

    - For 2.4 kernels, run:

      ```
      mount -t ext3 {dev} /mnt/mds
      ```

    - For 2.6 kernels, run:

      ```
      mount -t ldiskfs {dev} /mnt/mds
      ```

12. **Change to the new file system mount point. Type:**

    ```
    cd /mnt/mds
    ```

13. **Restore the file system backup. Type:**

    ```
    tar xzvpf {backup file}
    ```

14. **Restore the file system EAs. Type:**

    ```
    setfattr --restore=ea.bak
    ```

15. **Remove the (now invalid) recovery logs. Type:**

    ```
    rm OBJECTS/* CATALOGS
    ```

Again, the restore of the EAs (described in Step 6) is not currently required for OST devices, but this may change in the future.

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 whole 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 IO 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.

How do I control multiple services on one node independently?

You can do this by assigning an OST (or MDS) to a specific group, often with a name that relates to the service itself (e.g. ost1a, ost1b, ...). In the lmc configuration script, put each OST into a separate group, use:

```
lmc --add ost --group <name> ...
```

When starting up each OST use:

```
lconf --group <name>  {--reformat,--cleanup,etc} foo.xml
```

to start up each one individually.

Unless a group is specified all of the services on the that node will be affected by the command.

Beginning with Lustre 1.4.4, managing individual services has been substantially simplified.

The group / select mechanics are gone, and you can operate purely on the basis of service names:

```
lconf --service <service> [--reformat --cleanup ...] foo.xml
```

For example, if you add the service ost1-home, type:

```
lmc --add ost --ost ost1-home ...
```

You can start it with:

```
lconf --service ost1-home foo.xml
```

As before, if you do not specify a service, all services configured for that node will be affected by your command.

What extra resources are required for automated failover?

To automate failover with Lustre, you need power management software, remote control power equipment, and cluster management software.

Power Management Software

PowerMan, by the Lawrence Livermore National Laboratory, is a tool that manipulates remote power control (RPC) devices from a central location. PowerMan natively supports several RPC varieties. Expect-like configurability simplifies the addition of new devices. For more information about PowerMan, go to:

`http://www.llnl.gov/linux/powerman.html`

Other power management software is available, but PowerMan is the best we have used so far, and the one with which we are most familiar.

Power Equipment

A multi-port, Ethernet-addressable RPC is relatively inexpensive. For recommended products, see the list of supported hardware on the PowerMan website.

If you can afford them, Linux Network ICEboxes are very good tools. They combine both remote power control and remote serial console in a single unit.

Cluster management software

There are two options for cluster management software that have been implemented successfully by Lustre customers. Both software options are open source and available free for download.

- **Heartbeat**

The Heartbeat program is one of the core components of the High-Availability Linux (Linux-HA) project. Heartbeat is highly-portable, and runs on every known Linux platform, as well as FreeBSD and Solaris.

For information, see: `http://linux-ha.org/heartbeat/`

To download, see: `http://linux-ha.org/download/`

- **Red Hat Cluster Manager (CluManager)**

Red Hat Cluster Manager allows administrators to connect separate systems (called members or nodes) together to create failover clusters that ensure application availability and data integrity under several failure conditions.

Administrators can use Red Hat Cluster Manager with database applications, file sharing services, web servers, and more.

Note – CluManager requires two 10M LUNs visible to each member of a failover group.

For more information, see:
http://www.redhat.com/docs/manuals/enterprise/RHEL-3-Manual/cluster-suite/

For more download, see:
http://ftp.redhat.com/pub/redhat/linux/enterprise/3/en/RHCS/i386/SRPMS/

In the future, we hope to publish more information and sample scripts to configure Heartbeat and CluManager with Lustre.

Is there a way to tell which OST is being used by a client process?

If a process is doing I/O to a file, use the lfs getstripe command to see the OST to which it is writing.

Using cat as an example, run:

```
$ cat > foo
```

While that is running, on another terminal, run:

```
$ readlink /proc/$(pidof cat)/fd/1
/barn/users/jacob/tmp/foo
```

You can also ls -l /proc/<pid>/fd/ to find open files using Lustre.

```
$ 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
```

The output shows that this 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 also works with connections to the MDS - just replace osc with mdc and ost with mds in the above command.

I need multiple SCSI LUNs per HBA - what is the best way to do this?

The packaged kernels are configured approximately the same as the upstream RedHat and SuSE packages.

Currently, RHEL does not enable CONFIG_SCSI_MULTI_LUN because it is said to causes problems with some SCSI hardware.

If you need to enable this, you must set 'option scsi_mod max_scsi_luns=xx' (xx is typically 128) in either modprobe.conf (2.6 kernel) or modules.conf (2.4 kernel).

Passing this option as a kernel boot argument (in grub.conf or lilo.conf) will not work unless the kernel is compiled with CONFIG_SCSI_MULT_LUN=y

Can I run Lustre in a heterogeneous environment (32-and 64-bit machines)?

As of Lustre v1.4.2, this is supported with different word sizes. It is also supported for clients with different endianness (for example, i368 and PPC).

One limitation is that the PAGE_SIZE on the client must be at least 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 i386 clients are running with ia64 servers, the ia64 kernel must be compiled with 4kB PAGE_SIZE.

How do I clean up a device with lctl?

How do I destroy this object using lctl based on the following information:

lctl > device_list

0 UP obdfilter ost003_s1 ost003_s1_UUID 3

1 UP ost OSS OSS_UUID 2

2 UP echo_client ost003_s1_client 2b98ad95-28a6-ebb2-10e4-46a3ceef9007

1. **Try:**

```
lconf --cleanup --force
```

2. **If that does not work, start lctl (if it is not running already). Then, starting with the highest-numbered device and working backward, clean up each device:**

```
root# lctl
lctl> cfg_device ost003_s1_client
lctl> cleanup force
lctl> detach
lctl> cfg_device OSS
lctl> cleanup force
lctl> detach
lctl> cfg_device ost003_s1
lctl> cleanup force
lctl> detach
```

At this point it should also be possible to unload the Lustre modules.

How to build and configure Infiniband support for Lustre

The distributed kernels do not yet include 3rd-party Infiniband modules. As a result, our Lustre packages can not include IB network drivers for Lustre either, however we do distribute the source code. You will need to build your Infiniband software stack against the supplied kernel, and then build new Lustre packages. If this is outside your realm of expertise, and you are a Lustre enterprise-support customer, we can help.

- Volatire

  To build Lustre with Voltaire Infiniband sources, add: --with-vib=<path-to-voltaire-sources> as an argument to the configure script.

  To configure Lustre, use: --nettype vib --nid <IPoIB address>

- OpenIB generation 1 / Mellanox Gold

  To build Lustre with OpenIB Infiniband sources, add --with-openib=<path_to_openib sources> as an argument to the configure script.

To configure Lustre, use: --nettype openib --nid <IPoIB address>

- Silverstorm

  A Silverstorm driver for Lustre is available.

- OpenIB 1.0

  An OpenIB 1.0 driver for Lustre is available.

Currently (v1.4.5) the Voltaire IB module (kvibnal) will _not work on the Altix system. This is due to hardware differences in the Altix system.

To build Silverstorm with Lustre, configure Lustre with:

```
--with-iib=<path to silverstorm sources>
```

Can the same Lustre file system be mounted at multiple mount points on the same client system?

Yes, this is perfectly safe.

How do I identify files affected by a missing OST?

If an OST is missing for any reason, you may need to know what files are affected.

The file system should still be operational, even though one OST is missing, so from any mounted client node it is possible to generate a list of files that reside on that OST.

In such situations, we recommend marking the missing OST as unavailable, so clients and the MDS do not time out trying to contact it. On mixed MDS/client nodes:

1. **Generate a list of devices and determine the OST's device number.**

   ```
   $ lctl dl
   ```

2. **Deactivate the OST (on the OSS at the MDS).**

   ```
   $ lctl --device <OST device name or number> deactivate
   ```

If the OST later becomes available it needs to be reactivated. Run:

```
$ lctl --device <OST device number> activate
```

Determine all files 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.

It is possible to read the valid parts of a striped file (if necessary):

```
$ dd if=filename of=new_filename bs=4k conv=sync,noerror
```

Otherwise, it is possible to delete these files with "unlink" or "munlink".

If you need to need to know specifically which parts of the file are missing data you first need to determine the file layout (striping pattern), which includes the index of the missing OST:

```
$ lfs getstripe -v {filename}
```

The following computation is used 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

Example: for a file with 2 stripes, stripe size = 1M, bad OST is index 0 you would have holes in your file at:

```
[(2*N + 0)*1M, (2*N + 0)*1M + 1M - 1], N = { 0, 1, 2, ...}
```

If the file system can't be mounted, there isn't anything currently that would parse metadata directly from an MDS. If the bad OST is definitely not starting, options for mounting the file system anyway are to provide a loop device OST in its place, or to replace it with a newly formatted OST. In that case the missing objects are created and will read as zero-filled.

How-To: New Lustre network configuration

Updating Lustre's network configuration during an upgrade to version 1.4.6.

Outline necessary changes to Lustre configuration for the new networking features in v. 1.4.6. Further details may be found in the Lustre manual excerpts found at:

https://wiki.clusterfs.com/cfs/intra/FrontPage?action=AttachFile&do=get&target=LustreManual.pdf

Backwards Compatibility

The 1.4.6 version of Lustre itself uses the same wire protocols as the previous release, but has a different network addressing scheme and a much simpler configuration for routing.

In single-network configurations, LNET can be configured to work with the 1.4.5 networking (portals) so that rolling upgrades can be performed on a cluster. See the 'portals_compatibility' parameter below.

When 'portals_compatibility' is enabled, old XML configuration files remain compatible. lconf automatically converts old-style network addresses to the new LNET style.

If a rolling upgrade is not required (that is, all clients and servers can be stopped at one time), then follow the standard procedure:

1. Shut down all clients and servers

2. Install new packages everywhere

3. Edit the Lustre configuration

4. Update the configuration on the MDS with 'lconf --write_conf'

5. Restart

New Network Addressing

A NID is a Lustre network address. Every node has one NID for each network to which it is attached.

The NID has the form <address>[@<network>], where the <address> is the network address and <network> is an identifier for the network. (network type + instance)

Examples:

First TCP network: 192.73.220.107@tcp0

Second TCP network: 10.10.1.50@tcp1

Elan: 2@elan

The "--nid '*' " syntax for the generic client is still valid.

Modules/modprobe.conf

Network hardware and routing are now configured via module parameters, specified in the usual locations. Depending on your kernel version and Linux distribution, this may be /etc/modules.conf,

/etc/modprobe.conf, or /etc/modprobe.conf.local.

All old Lustre configuration lines should be removed from the module configuration file. The RPM install should do this, but check to be certain.

The base module configuration requires two lines:

```
alias lustre llite
options lnet networks=tcp0
```

A full list of options can be found at Module Parameters on page 37. Detailed examples can be found in the section, 'Configuring the Lustre Network'. Some brief examples:

Example 1: Use eth1 instead of eth0:

options lnet networks="tcp0(eth1)"

Example 2: Servers have two tcp networks and one Elan network. Clients are either TCP or Elan.

Servers: options lnet 'networks="tcp0(eth0,eth1),elan0"

Elan clients: options lnet networks=elan0

TCP clients: options lnet networks=tcp0

Portals Compatibility

If you are upgrading Lustre on all clients and servers at the same time, then you may skip this section.

If you need to keep the file system running while some clients are upgraded, the following module parameter controls interoperability with pre-1.4.6 Lustre.

Compatibility between versions is not possible if you are using portals routers/gateways. If you use gateways, you must update the clients, gateways, and servers at the same time.

portals_compatibility="strong"|"weak"|"none"

"strong" is compatible with Lustre 1.4.5, and 1.4.6 running in either 'strong' or 'weak' compatibility mode.

Since this is the only mode compatible with 1.4.5, all 1.4.6 nodes in the cluster must use "strong" until the last 1.4.5 node has been upgraded.

"weak" is not compatible with 1.4.5, or with 1.4.6 running in "none" mode.

"none" is not compatible with 1.4.5, or with 1.4.6 running in 'strong' mode.

For more information, see Upgrading Lustre on page 117.

Note – Lustre v.1.4.2 through v.1.4.5 clients are only compatible zero-conf mounting from a 1.4.6 MDS if the MDS was originally formatted with Lustre 1.4.5 or earlier. If the file system was formatted with v.1.4.6 on the MDS, or "lconf --write-conf" was run on the MDS then the backward compatibility is lost. It is still possible to mount 1.4.2 through 1.4.5 clients with "lconf --node {client_node} {config}.xml".

How to fix bad LAST_ID on an OST

The file system must be stopped on all servers prior to performing this procedure.

For hex <–> decimal translations:

Use GDB:

```
(gdb) p /x 15028
$2 = 0x3ab4
```

Or bc:

```
echo "obase=16; 15028" | bc
```

1. Determine a reasonable value for LAST_ID. Check on the MDS:

```
# mount -t ldiskfs /dev/<mdsdev> /mnt/mds
# od -Ax -td8 /mnt/mds/lov_objid
```

There is one entry for each OST, in OST index order. This is what the MDS thinks the last in-use object is.

2. Determine the OST index for this OST.

```
# od -Ax -td4 /mnt/ost/last_rcvd
```

It will have it at offset 0x8c.

3. Check on the OST. With debugfs, check LAST_ID:

```
debugfs -c -R 'dump /O/0/LAST_ID /tmp/LAST_ID' /dev/XXX ; od -Ax -td8
/tmp/LAST_ID"
```

4. Check objects on the OST:

```
mount -rt ldiskfs /dev/{ostdev} /mnt/ost
# note the ls below is a number one and not a letter L
ls -1s /mnt/ost/O/0/d* | grep -v [a-z] |
sort -k2 -n > /tmp/objects.{diskname}

tail -30 /tmp/objects.{diskname}
```

This shows you the OST state. There may be some pre-created orphans, check for zero-length objects. Any zero-length objects with IDs higher than LAST_ID should be deleted. New objects will be pre-created.

If the OST LAST_ID value matches that for the objects existing on the OST, then it is possible the lov_objid file on the MDS is incorrect. Delete the lov_objid file on the MDS and it will be re-created from the LAST_ID on the OSTs.

If you determine the LAST_ID file on the OST is incorrect (that is, it does not match what objects exist, does not match the MDS lov_objid value), then you have decided on a proper value for LAST_ID.

Once you have decided on a proper value for LAST_ID, use this repair procedure.

1. **Access:**

```
mount -t ldiskfs /dev/{ostdev} /mnt/ost
```

2. **Check the current:**

```
od -Ax -td8 /mnt/ost/O/0/LAST_ID
```

3. **Be very safe, only work on backups:**

```
cp /mnt/ost/O/0/LAST_ID /tmp/LAST_ID
```

4. **Convert binary to text:**

```
xxd /tmp/LAST_ID /tmp/LAST_ID.asc
```

5. **Fix:**

```
vi /tmp/LAST_ID.asc
```

6. **Convert to binary:**

```
xxd -r /tmp/LAST_ID.asc /tmp/LAST_ID.new
```

7. **Verify:**

```
od -Ax -td8 /tmp/LAST_ID.new
```

8. **Replace:**

```
cp /tmp/LAST_ID.new /mnt/ost/O/0/LAST_ID
```

9. **Clean up:**

```
umount /mnt/ost
```

Why can't I run an OST and a client on the same machine?

Consider the case of a "client" with dirty file system pages in memory and memory pressure. A kernel thread is woken to flush dirty pages to the file system, and it writes to local OST. The OST needs to do an allocation in order to complete the write. The allocation is blocked, waiting for the above kernel thread to complete the write and free up some memory. This is a deadlock.

Also, if the node with both a client and OST crash, then the OST waits, during recovery, for the client that was mounted on that node to recover. However, since the client crashed, it is considered a new client to the OST, and is blocked from mounting until recovery completes. As a result, this is currently considered a double failure and recovery cannot complete successfully.

Information on the Socket LND (socklnd) protocol

Lustre layers the socket LND (socklnd) protocol above TCP/IP. The first message sent on the TCP/IP bytestream is HELLO, which is used to negotiate connection attributes. The protocol version is determined by looking at the first 4+4 bytes of the hello message, which contain a magic number and the protocol version

In KSOCK_PROTO_V1, the hello message is an lnet_hdr_t of type LNET_MSG_HELLO, with the dest_nid (Destination Server/Machine) replaced by net_magicversion_t. This is followed by 'payload_length' bytes of IP addresses (each 4 bytes) which list the interfaces that the sending socklnd owns. The whole message is sent in little-endian (LE) byte order. There is no socklnd level V1 protocol after the initial HELLO meaning everything that follows is unencapsulated LNET messages.

In KSOCK_PROTO_V2, the hello message is a ksock_hello_msg_t. The whole message is sent in byte order of sender and the bytesex of 'kshm_magic' is used on arrival to determine if the receiver needs to flip. From then on, every message is a ksock_msg_t also sent in the byte order by sender. This either encapsulates an LNET message (ksm_type == KSOCK_MSG_LNET) or is a NOOP. Every message includes zero-copy request and ACK cookies in every message so that a zero-copy sender can determine when the source buffer can be released without resorting to a kernel patch. The NOOP is provided for delivering a zero-copy ACK when there is no LNET message to back it on.

Note that socklnd may connect to its peers via a "bundle" of sockets - one for bidirectional "ping-pong" data and the other two for unidirectional bulk data. However the message protocol on every socket is as described earlier.

Information on the Lustre Networking (LNET) protocol

Lustre layers the socket LND (socklnd) protocol above TCP/IP. Every LNET message is an lnet_hdr_t sent in (little-endian (LE) byte order followed by 'payload_length' bytes of opaque payload data. There are four types of messages.

- PUT - request to send data contained in the payload
- ACK - response to a PUT with ack_wmd != LNET_WIRE_HANDLE_NONE
- GET - request to fetch data
- REPLY - response to a GET with data in the payload

Typically, ACK and GET messages have 0 bytes of payload.

Explanation of: '... previously skipped # similar messages' in Lustre logs

Unlike syslog, which occupies exactly identical lines, the space for Lustre messages is occupied if there are bursts of messages from the same line of code, even if they are not sequential. This avoids duplication of the same event from different clients, or in cases where two or more messages are repeated.

All messages are kept in the Lustre kernel debug log, so "lctl dk" at that time would show all messages (in case they are not wrapped).

Printing a large number of messages to the kernel console can dramatically slow down the system. As this happens with IRQs disabled and for a slow console, it severely impacts overall system performance when there are large number of messages.

For example:

```
LustreError: 559:0:(genops.c:1292:obd_export_evict_by_nid())
evicting
b155f37b-b426-ccc2-f0a9-bfbf00000000 at adminstrative request
LustreError: 559:0:(genops.c:1292:obd_export_evict_by_nid())
previously skipped 2 similar messages
```

In this case, the 'similar' messages are reported for the exact line of source, without matching the text. Therefore, this is expected output for evictions of more than one client.

What should I do if I suspect device corruption (Example: disk errors)

Keep these points in mind when trying to recover from device-induced corruption.

- Stop using the device as soon as possible (if you have a choice).

  The longer corruption is present on a device, the greater the risk that it will cause further corruption. Normally, ext3 marks the file system read-only if any corruption is detected or if there are I/O errors when reading or writing metadata to the file system. This can only be cleared by shutting down Lustre on the device (use --force or reboot if necessary).

- Proceed carefully

  If you take incorrect action, you can make an otherwise-recoverable situation worse. ext3 has very robust metadata formats and can often recover a large amount of data even when a significant portion of the device is bad.

- Keep a log of all actions and output in a safe place.

  If you perform multiple file system checks and/or actions to repair the file system, save all logs. They may provide valuable insight into problems encountered.

  Normally, the first thing to do is a read-only file system check, after the Lustre service (MDS or OST) has been stopped. If it is not possible to stop the service, you can run a read-only file system check when the device is in use. If running a file system check while the device is in use, e2fsck cannot always coordinate data gathered at the start of the run with data gathered later in the run and will report incorrect file system errors. The number of errors is dependent upon the length of check (approximately equal to the device size) and the load on the file system. In this situation, you should run e2fsck multiple times on the device and look for errors that are persistent across runs, and ignore transient errors.

  To run a read-only file system check, we recommend that you use the latest e2fsck, available at http://www.sun.com/software/products/lustre/get.jsp.

  On the system with the suspected bad device (in the example below, /dev/sda is used), run:

```
[root@mds]# script /root/e2fsck-1.sda
Script started, file is /root/e2fsck-1.sda
[root@mds]# e2fsck -fn /dev/sda
e2fsck 1.35-lfck8 (05-Feb-2005)
Warning: skipping journal recovery because doing a read-only
filesystem check
Pass 1: Checking inodes, blocks, and sizes
[root@mds]# exit
Script done, file is /tmp/foo
```

In many cases, the extent of corruption is small (some unlinked files or directories, or perhaps some parts of an inode table have been wiped out). If there are serious file system problems, e2fsck may need to use a backup superblock (reports if it does). This causes all of the "group summary" information to be incorrect. In and of itself, this is not a serious error as this information is redundant and e2fsck can reconstruct this data. If the primary superblock is not valid, then there is some corruption at the start of the device and some amount of data may be lost. The data is somewhat protected from beginning-of-device corruption (which is one of the more common cases) because of the large journal placed at the start of the file system.

The amount of time taken to run such a check is usually 4 hours for a 1 TB MDS device or a 2 TB OST device, but varies with the number of files and the amount of data in the file system. If there are severe problems with the file system, it can take 8-12 hours to complete the check.

Depending on the type of corruption, it is sometimes helpful to use debugfs to examine the file system directly and learn more about the corruption.

```
[root@mds]# script /root/debugfs.sda
[root@mds]# debugfs /dev/sda
```

debugfs 1.35-lfsk8 (05-Feb-2005)

```
debugfs> stats
```

{shows superblock and group summary information}

```
debugfs> ls
```

{shows directory listing}

```
debugfs> stat <inum>
```

{shows inode information for inode number <inum>}

```
debugfs> stat name
```

{shows inode information for inode "name"}

```
debugfs> cd dir
```

{change into directory "dir", "ROOT" is start of Lustre-visible namespace}

```
debugfs> quit
```

Once you have assessed the damage (possibly with the assistance of Lustre Support, depending on the nature of the corruption), then fixing it is the next step. Often, it is prudent to make a backup of the file system metadata (time and space permitting) in case there is a problem or if it is unclear whether e2fsck will make the correct action (in most cases it will). To make a metadata backup, run:

```
[root@mds]# e2image /dev/sda /bigplace/sda.e2image
```

In most cases, running `e2fsck -fp $device` will fix most types of corruption. The e2fsck program has been used for many years and has been tested with a huge number of different corruption scenarios. If you suspect serious corruption, or do not expect e2fsck to fix the problem, then consider running a manual check, `e2fsck -f $device`. The limitation of the manual check is that it is interactive and can be quite lengthy if there are a lot of problems.

How do I clean up a device with lctl?

1. **Run:**

```
lconf --cleanup --force
```

2. **If that does not work, then start lctl (if it is not already started).**

3. **Then starting with the highest-numbered device and working backward, clean up each device.**

```
root# clctl
lctl> cfg_device ost003_s1_client
lctl> cleanup force
lctl> detach
lctl> cfg_device OSS
lctl> cleanup force
lctl> detach
lctl> cfg_device ost003_s1
lctl> cleanup force
lctl> detach
```

At this point, you should be possible to unload the Lustre modules.

What is the default block size for Lustre?

The on-disk block size for Lustre is 4 KB (same as ext3). Nevertheless, Lustre goes to great lengths to do 1 MB reads and writes to the disk, as large requests are a key to getting very high performance.

How do I determine which Lustre server (MDS/OST) was connected to a particular storage device

In instances when the hardware configuration has changed (e.g., moving equipment and re-connecting it), it is important to connect the right storage devices to the associated Lustre servers.

Lustre writes a UUID to every OST and MDS. To view this information:

1. **Mount the storage device as ldiskfs**

```
mount -t ldiskfs /dev/foo /mnt/tmp
```

2. **Inspect the contents of the last_rcvd file in the root directory**

```
strings /mnt/tmp/last_rcvd
```

The MDS/OST UUID is the first element in the last_rcvd file and is in a human readable form (e.g. mds1_UUID).

3. **Unmount the storage device and connect it to the appropriate Lustre server.**

```
umount /mnt/tmp
```

Note – It is not possible to mismatch storage devices with their Lustre servers. If Lustre tries to mount such devices incorrectly, it would report a UUID mismatch to the syslog and refuse to mount.

Does the mount option "--bind" allow mounting a Lustre file system to multiple directories on the same client system?

Yes, this is supported. In fact, it is entirely handled by the VFS. No special file system support is required.

What operations take place in Lustre when a new file is created?

This is a high level description of what operations take place in Lustre when a new file is created. It corresponds to Lustre version 1.4.5.

- On the Lustre client:

1. **Create (/path/file, mode).**
2. **For every component in path, execute IT_LOOKUP intent (LDLM_ENQUEUE RPC) to MDS.**
3. **Execute IT_OPEN intent (LDLM_ENQUEUE RPC) to MDS.**

- On the MDS:

1. **Lock the parent directory.**
2. **Create the file.**
3. **Setattr on the file to set desired owner/mode.**
4. **Setattr on parent to update ATIME/CTIME.**
5. **Determine the default striping pattern.**
6. **Set the file's extended attribute to the desired stripping pattern.**
7. **For every OST that this file will have stripes on, see if there is a spare.**
8. **Assign precreated objects (if any) to the file.**
9. **Update the extended attribute holding OST oids.**
10. **Reply to client with no lock in reply.**

- On the journal:

  ext3 journaling is asynchronous unless a handle specifically requests a synchronous operation. file system-modifying operations on the MDS that make up a single file create operation are:

  - Allocate inode (inode bitmap, group descriptor, new inode)
  - Create directory entry (directory block, parent inode for timestamps)
  - Update lov_objids file (Lustre file)
  - Update last_rcvd file (Lustre file)

  For a single inode, each of the above items dirties a single block in the journal (7 blocks = 28 KB in total). When many new files are created at one time, dirty blocks are merged in the journal, because each block needs to be dirtied only once per transaction (5s or 1/4 of full journal, whichever occurs earlier). For 1,000 files created in a single directory, this works out to 516 KB, if they are all created within the same transaction.

  In 2.6 kernels it is possible to tune the ext3 journal commit interval with "-o commit={seconds}". This may be desirable for performance testing.

  ext3 code reserves a lot more blocks (about 70) for worst-case scenarios (e.g., growing a directory which also results in a split of the directory index, quota updates, adding new indirect blocks for each of the Lustre files modified). These are returned to the journal when the transaction is complete; most are returned unused.

  To avoid spurious journal commits due to these temporary reservations, calculate the journal size based on this formula (assuming a default of 32 MDS threads):

  70 blocks/thread * 32 threads * 4 KB/block * 4 = 35840 KB

What is the Lustre data path?

On the OST, data is read directly from the disk into pre-allocated network I/O buffers, in chunks up to 1 MB in size. This data is sent (zero-copy where possible) to the clients, where it is put (again, zero-copy where possible) into the file's data mapping. The clients maintain local writeback and readahead caches for Lustre.

On the OST, the file system metadata such as inodes, bitmaps and file allocation information is cached in RAM (up to the maximum amount that the kernel allows). No user data is currently cached on the OST.

In cases where only few files are read by many clients, it makes sense to use a RAID device with a lot of local RAM cache so that the multiple read requests can skip the disk access.

The networking code bundles up page requests into a maximum of 1 MB in a single RPC to minimize overhead. In each client OSC, this is controlled by the `/proc/fs/lustre/osc/*/max_pages_per_rpc` field. The size of the writeback cache can be tuned via `/proc/fs/lustre/osc/*/max_dirty_mb`. The size of the readahead can be tuned via `/proc/fs/lustre/llite/max_read_ahead_mb`. Total client side cache usage can be limited via `/proc/fs/lustre/llite/max_cached_mb`.

Questions about using Lustre quotas

This section covers various aspects of using Lustre quotas.

When I enable quotas with lfs quotaon, will it automatically set default quotas for all users or do I have to set them for each user/group individually?

In that case, the default limit will be 0, which means no limit.

What happens if a user/group has already more files/disk usage than his quotas allows?

Given that it will be 0 initially, no users will be over quotas. To preempt the next question, if a user has a limit set that is less than his existing usage, he will simply start to get -EDQUOT errors on subsequent attempts to write data.

We only want group quotas, do we have to enable user quotas as well?

We do not know of any particular failure if only group quotas are enabled, but the more your use cases match our testing then the better off you will be.

For user quotas, even if you do not want to enforce limits, you can enable quotas but not set any limits. Doing this makes future operation of enabling limits on users easier (when/if you decide to) as usage will already be tracked and accounted for (saving you the need to do that initial accounting). It also provides you with a means to quickly assess how much space is being consumed on a user-by-user basis.

When mounting an MDT filesystm, the kernel crashes. What do I do?

On Lustre versions prior to 1.6.5, use this procedure:

1. **Try to mount the file system with o abort_recovery as an option.**
2. **If this does not work, try to mount the file system as -t ldiskfs.**

```
mount -t ldiskfs
```

3. **If that works, try to truncate the last_rcvd file.**

```
mount -t ldiskfs /dev/MDSDEV /mnt/mds
cp /mnt/mds/last_rcvd /mnt/mds/last_rcvd.sav
cp /mnt/mds/last_rcvd /tmp/last_rcvd.sav
dd if=/mnt/mds/last_rcvd.sav of=/mnt/mds/last_rcvd bs=8k count=1
umount /mnt/mds

mount -t lustre /dev/MSDDEV /mnt/mds
```

Lustre version 1.6.5 and later should not encounter this problem.

How do I determine which Ethernet interfaces Lustre uses?

Use the `lctl list_nids` command to show the interfaces that Lustre is using. Keep in mind that when socklnd bonding is used (e.g., networks="tcp0(eth0,eth1)"), the LNET NID only picks up the IP address of the first interface in the network's specification (e.g., the IP address of eth0@tcp), despite LNET trying to make use of both interfaces.

Moreover, the Ethernet interface in use is solely determined by the Linux IP routing. For example, if you have two Ethernet interfaces (eth0 and eth1), and you direct LNET to use eth0 only (e.g. networks="tcp(eth0)"), traffic can still use eth1 if Linux IP routing selects it because of misconfigured routing (both interfaces are in the same IP network, the routing table entry for eth1 comes first or by mistake).

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.

CMOBD Cache Management OBD. A special device which implements remote cache flushed and migration among devices.

COBD Caching OBD. A driver which decides when to use a proxy or a locally-running cache and when to go to a master server. Formerly, this abbreviation was used for the term 'collaborative cache'.

Collaborative Cache A read cache instantiated on nodes that can be clients or dedicated systems. It enables client-to-client data transfer, thereby enabling enormous scalability benefits for mostly read-only situations. A collaborative cache is not currently implemented in Lustre.

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.

GNS Global Namespace. A GNS enables clients to access files without knowing their location. It also enables an administrator to aggregate file storage across distributed storage devices and manage it as a single file system.

Group Lock

Group upcall

GSS Group Sweeping Scheduling. A disk sched uling strategy in which requests are served in cycles, in a round-robin manner.

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.

J

Join File

K

Kerberos An authentication mechanism, optionally available in Lustre 1.8 as a GSS backend.

L

LAID Lustre RAID. A mechanism whereby the LOV stripes I/O over a number of OSTs with redundancy. This functionality is expected to be introduced in Lustre 2.0.

LBUG A bug that Lustre writes into a log indicating a serious system failure.

LDLM Lustre Distributed Lock Manager.

lfind A subcommand of lfs to find inodes associated with objects.

lfs A Lustre file system utility named after fs (AFS), cfs (CODA), and lfs (Intermezzo).

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.

LNETrpc An RPC protocol layered on LNET. This protocol deals with stateful servers and has exactly-once semantics and built in support for recovery.

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 An operating instance with a mounted Lustre file system.

MDC An operating instance with a mounted Lustre file system.

MDD An operating instance with a mounted Lustre file system.

MDS An operating instance with a mounted Lustre file system.

MDS client Same as MDC.

MDS server Same as MDS.

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 ave not yet been flushed to a storage device or server. InterMezzo is one of the first network file systems to have a metadata write-back cache.

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.

Mount object

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.

OBDFS Object Based File System. A now obsolete single node object file system that stores data and metadata on object devices.

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 system that runs an object storage service software stack.

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 A group of OSTs can be combined into a pool with unique access permissions and stripe characteristics. Each OST is a member of only one pool, while an MDT can serve files from multiple pools. A client accesses one pool on the the file system; the MDT stores files from / for that client only on that pool's OSTs.

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 older term for LNETrpc.

R

Raw operations VFS operations introduced by Lustre to implement operations such as mkdir, rmdir, link, rename with a single RPC to the server. Other file systems would typically use more operations. The expense of the raw operation is omitting the update of client namespace caches after obtaining a successful result.

Remote user handling

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.

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.

Z

zeroconf A method to start a client without an XML file. The mount command gets a client startup llog from a specified MDS. This is an obsolete method in Lustre 1.6 and later.

Index

M

N

O

P

Q

R

S

T

U

V

W