



Lustre™ 1.6 Operations Manual

Sun Microsystems, Inc.
www.sun.com

Part No. 820-3681-10

Lustre manual version: Lustre_1.6_man_v1.11

March 2008

Copyright© 2008 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A. All rights reserved.

U.S. Government Rights - Commercial software. Government users are subject to the Sun Microsystems, Inc. standard license agreement and applicable provisions of the FAR and its supplements.

Sun, Sun Microsystems, the Sun logo and Lustre are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

Products covered by and information contained in this service manual are controlled by U.S. Export Control laws and may be subject to the export or import laws in other countries. Nuclear, missile, chemical biological weapons or nuclear maritime end uses or end users, whether direct or indirect, are strictly prohibited. Export or reexport to countries subject to U.S. embargo or to entities identified on U.S. export exclusion lists, including, but not limited to, the denied persons and specially designated nationals lists is strictly prohibited.

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.



Please
Recycle



Adobe PostScript



Please
Recycle



Adobe PostScript

Contents

Part I Lustre Architecture

1. A Cluster with Lustre 1-1

1.1 What is Lustre? 1-1

1.2 Lustre Software 1-2

1.3 Lustre Components 1-4

1.3.1 MGS 1-5

1.3.2 MDT 1-6

1.3.3 OSTs 1-6

1.3.4 Lustre Client Nodes 1-6

1.3.5 LNET 1-7

2. Understanding Lustre Networking 2-1

2.1 Introduction to LNET 2-1

2.2 Supported Network Types 2-2

2.3 Important Terms 2-2

Part II Lustre Administration

3. Prerequisites 3-1

- 3.1 Preparing to Install Lustre 3-1
 - 3.1.1 How to Get Lustre 3-2
 - 3.1.2 Supported Configurations 3-3
- 3.2 Using a Pre-Packaged Lustre Release 3-4
 - 3.2.1 Choosing a Pre-Packaged Kernel 3-4
 - 3.2.2 Lustre Tools 3-5
 - 3.2.3 Other Required Software 3-5
- 3.3 Environmental Requirements 3-6
 - 3.3.1 SSH Access 3-6
 - 3.3.2 Consistent Clocks 3-7
 - 3.3.3 Universal UID/GID 3-7
 - 3.3.4 Choosing a Proper Kernel I/O Scheduler 3-7
 - 3.3.5 Changing the I/O Scheduler 3-8
- 3.4 Memory Requirements 3-9
 - 3.4.1 Determining the MDS's Memory 3-9
 - 3.4.2 OSS Memory Requirements 3-10

4.	Lustre Installation	4-1
4.1	Installing Lustre	4-2
4.1.1	MountConf	4-2
4.2	Quick Configuration of Lustre	4-3
4.2.1	Simple Configurations	4-4
4.2.2	More Complex Configurations	4-9
4.2.3	Other Configuration Tasks	4-12
4.3	Building from Source	4-14
4.3.1	Building Your Own Kernel	4-14
4.3.2	Building Lustre	4-16
4.3.3	Building from Source	4-20
4.4	Building a Lustre Source Tarball	4-21
4.4.1	Lustre Source Tarball from Lustre Source RPM	4-21
4.4.2	Lustre Source Tarball from CVS	4-22
5.	Configuring the Lustre Network	5-1
5.1	Designing Your Lustre Network	5-1
5.1.1	Identify All Lustre Networks	5-1
5.1.2	Identify Nodes to Route Between Networks	5-1
5.1.3	Identify Network Interfaces to Include/Exclude from LNET	5-2
5.1.4	Determine Cluster-wide Module Configuration	5-2
5.1.5	Determine Appropriate Mount Parameters for Clients	5-3
5.2	Configuring Your Lustre Network	5-4
5.2.1	Module Parameters	5-4
5.2.2	Module Parameters - Routing	5-7
5.2.3	Downed Routers	5-9
5.3	Starting and Stopping LNET	5-10
5.3.1	Starting LNET	5-10
5.3.2	Stopping LNET	5-11

6. Configuring Lustre - Examples 6-1

6.1 Simple TCP Network 6-1

6.1.1 Lustre with Combined MGS/MDT 6-1

6.1.2 Lustre with Separate MGS and MDT 6-3

7. More Complicated Configurations 7-1

7.1 Multihomed Servers 7-1

7.1.1 Modprobe.conf 7-1

7.1.2 Start Servers 7-3

7.1.3 Start Clients 7-4

7.2 Elan to TCP Routing 7-4

7.2.1 Modprobe.conf 7-4

7.2.2 Start servers 7-5

7.2.3 Start clients 7-5

7.3 Load Balancing with Infiniband 7-5

7.3.1 Modprobe.conf 7-5

7.3.2 Start servers 7-6

7.3.3 Start clients 7-6

8. Failover 8-1

- 8.1 What is Failover? 8-1
 - 8.1.1 The Power Management Software 8-3
 - 8.1.2 Power Equipment 8-3
 - 8.1.3 Heartbeat 8-3
 - 8.1.4 Connection Handling During Failover 8-4
 - 8.1.5 Roles of Nodes in a Failover 8-5
- 8.2 OST Failover Review 8-5
- 8.3 MDS Failover Review 8-6
- 8.4 Configuring MDS and OSTs for Failover 8-6
 - 8.4.1 Starting/Stopping a Resource 8-6
 - 8.4.2 Active/Active Failover Configuration 8-6
 - 8.4.3 Hardware Requirements for Failover 8-7
- 8.5 Setting Up Failover with Heartbeat V1 8-8
 - 8.5.1 Installing the Software 8-8
 - 8.5.2 Mon (Status Monitor) 8-15
- 8.6 Using MMP 8-18
- 8.7 Setting Up Failover with Heartbeat V2 8-19
 - 8.7.1 Installing the Software 8-19
 - 8.7.2 Configuring the Hardware 8-20
 - 8.7.3 Operation 8-23
- 8.8 Considerations with Failover Software and Solutions 8-24

9. Configuring Quotas 9-1

- 9.1 Working with Quotas 9-1
 - 9.1.1 Configuring Disk Quotas 9-2
 - 9.1.2 Creating Quota Files and Quota Administration 9-3
 - 9.1.3 Resetting the Quota 9-4
 - 9.1.4 Quota Allocation 9-5

10. RAID 10-1

- 10.1 Considerations for Backend Storage 10-1
 - 10.1.1 Reliability 10-2
 - 10.1.2 Selecting Storage for the MDS and OSS 10-2
 - 10.1.3 Understanding Double Failures with Software and Hardware RAID5 10-3
 - 10.1.4 Performance Considerations 10-3
 - 10.1.5 Formatting 10-4
- 10.2 Insights into Disk Performance Measurement 10-4
 - 10.2.1 Sample Graphs 10-8
- 10.3 Creating an External Journal 10-12

11. Kerberos 11-1

- 11.1 What is Kerberos? 11-1
- 11.2 Lustre Setup with Kerberos 11-2
 - 11.2.1 Configuring Kerberos for Lustre 11-2
 - 11.2.2 Types of Lustre-Kerberos Flavors 11-9

12. Bonding 12-1

- 12.1 Network Bonding 12-1
- 12.2 Requirements 12-2
- 12.3 Using Lustre with Multiple NICs versus Bonding NICs 12-4
- 12.4 Bonding Module Parameters 12-5
- 12.5 Setting Up Bonding 12-5
 - 12.5.1 Examples 12-9
- 12.6 Configuring Lustre with Bonding 12-11
 - 12.6.1 Bonding References 12-11

- 13. Upgrading Lustre 13-1**
 - 13.1 Lustre Interoperability 13-1
 - 13.2 Upgrading from Version 1.4.11 to Version 1.6.3 13-2
 - 13.2.1 Upgrade Requirements 13-2
 - 13.2.2 Supported Upgrade Paths 13-3
 - 13.2.3 Starting Clients 13-4
 - 13.2.4 Upgrading a Single Filesystem 13-4
 - 13.2.5 Upgrading Multiple Filesystems with a Shared MGS 13-7
 - 13.3 Downgrading Lustre from Version 1.6.3 to Version 1.4.11 13-8
 - 13.3.1 Downgrade Requirements 13-8
 - 13.3.2 Downgrading a Filesystem 13-9
- 14. Lustre SNMP Module 14-1**
 - 14.1 Installing the Lustre SNMP Module 14-2
 - 14.2 Building the Lustre SNMP Module 14-2
 - 14.3 Using the Lustre SNMP Module 14-3
- 15. Backup and Restore 15-1**
 - 15.1 Lustre Backups 15-1
 - 15.1.1 Client Filesystem-level Backups 15-1
 - 15.1.2 Performing Device-level Backups 15-2
 - 15.1.3 Performing File-level Backups 15-2
 - 15.2 Restoring from a File-level Backup 15-4
- 16. POSIX 16-1**
 - 16.1 Installing POSIX 16-2
 - 16.2 Running POSIX Tests Against Lustre 16-4
 - 16.3 Isolating and Debugging Failures 16-5

17. Benchmarking 17-1

17.1 Bonnie++ Benchmark 17-2

17.2 IOR Benchmark 17-3

17.3 IOzone Benchmark 17-5

18. Lustre Recovery 18-1

18.1 Recovering Lustre 18-1

18.2 Types of Failure 18-2

18.2.1 Client Failure 18-2

18.2.2 MDS Failure (and Failover) 18-2

18.2.3 OST Failure 18-3

18.2.4 Network Partition 18-3

Part III Lustre Tuning, Monitoring and Troubleshooting

19. Lustre I/O Kit 19-1

19.1 Lustre I/O Kit Description and Prerequisites 19-1

19.1.1 Downloading an I/O Kit 19-2

19.1.2 Prerequisites to Using an I/O Kit 19-2

19.2 Running I/O Kit Tests 19-2

19.2.1 sgpdd_survey 19-3

19.2.2 obdfilter_survey 19-4

19.2.3 ost_survey 19-9

19.3 PIOS Test Tool 19-10

19.3.1 Synopsis 19-11

19.3.2 PIOS I/O Modes 19-12

19.3.3 PIOS Parameters 19-13

19.3.4 PIOS Examples 19-16

19.4	LNET Self-Test	19-17
19.4.1	Introduction to LNET Self-Test	19-17
19.4.2	LNET Self-Test Concepts	19-19
19.4.3	LNET Self-Test Commands	19-20
20.	LustreProc	20-1
20.1	Introduction	20-1
20.1.1	/proc Entries for Lustre	20-2
20.2	Lustre I/O Tunables	20-7
20.2.1	Client I/O RPC Stream Tunables	20-7
20.2.2	Watching the Client RPC Stream	20-9
20.2.3	Client Read-Write Offset Survey	20-10
20.2.4	Client Read-Write Extents Survey	20-11
20.2.5	Watching the OST Block I/O Stream	20-12
20.2.6	Mechanics of Lustre Readahead	20-13
20.2.7	mballoc History	20-14
20.2.8	mballoc3 Tunables	20-16
20.2.9	Locking	20-18
20.3	Debug Support	20-19
20.3.1	RPC Information for Other OBD Devices	20-20

21.	Lustre Tuning	21-1
21.1	Module Options	21-1
21.1.1	MDS Threads	21-2
21.2	LNET Tunables	21-3
21.3	Options to Format MDT and OST Filesystems	21-4
21.3.1	Planning for Inodes	21-4
21.3.2	Calculating MDT Size	21-4
21.3.3	Overriding Default Formatting Options	21-5
21.4	Network Tuning	21-6
21.5	DDN Tuning	21-6
21.5.1	Setting Readahead and MF	21-7
21.5.2	Setting Segment Size	21-7
21.5.3	Setting Write-Back Cache	21-8
21.5.4	Setting maxcmds	21-8
21.5.5	Further Tuning Tips	21-9
21.6	Large-Scale Tuning for Cray XT and Equivalents	21-10
21.6.1	Network Tunables	21-11
21.7	Lockless I/O Tunables	21-12

22.	Lustre Troubleshooting Tips	22-1
22.1	Lustre Error Messages and Logs	22-1
22.1.1	Lustre Error Messages	22-1
22.1.2	Lustre Logs	22-1
22.2	Lustre Performance Tips	22-2
22.2.1	Setting SCSI I/O Sizes	22-2
22.2.2	Write Performance Better Than Read Performance	22-2
22.2.3	OST Object is Missing or Damaged	22-3
22.2.4	OSTs Become Read-Only	22-4
22.2.5	Identifying a Missing OST	22-4
22.2.6	Changing Parameters	22-6
22.2.7	Default Striping	22-7
22.2.8	Erasing a Filesystem	22-7
22.2.9	Reclaiming Reserved Disk Space	22-8
22.2.10	Considerations in Connecting a SAN with Lustre	22-8
22.2.11	Handling/Debugging "Bind: Address already in use" Error	22-8
22.2.12	Replacing An Existing OST or MDS	22-9
22.2.13	Handling/Debugging Error "- 28"	22-10
22.2.14	Triggering Watchdog for PID NNN	22-10
22.2.15	Handling Timeouts on Initial Lustre Setup	22-11
22.2.16	Handling/Debugging "LustreError: xxx went back in time"	22-11
22.2.17	Lustre Error: "Slow Start_Page_Write"	22-12
22.2.18	Drawbacks in Doing Multi-client O_APPEND Writes	22-12
22.2.19	Slowdown Occurs During Lustre Startup	22-13
22.2.20	Log Message 'Out of Memory' on OST	22-13

23.	Lustre Debugging	23-1
23.1	Lustre Debug Messages	23-2
23.1.1	Format of Lustre Debug Messages	23-3
23.2	Tools for Lustre Debugging	23-4
23.2.1	Lustre Debug Daemon	23-5
23.2.2	Controlling the Kernel Debug Log	23-6
23.2.3	The lctl Tool	23-7
23.2.4	Finding Memory Leaks	23-8
23.2.5	Printing to /var/log/messages	23-9
23.2.6	Tracing Lock Traffic	23-9
23.2.7	Sample lctl Run	23-9
23.2.8	Adding Debugging to the Lustre Source Code	23-10
23.2.9	Debugging in UML	23-10
23.3	Using Strace for Troubleshooting	23-11
23.4	Looking at Disk Content	23-12
23.4.1	Determine the Lustre UUID of an OST	23-13
23.4.2	Tcpdump	23-13
23.5	Ptlrpc Request History	23-14
23.6	LWT Tracing	23-15

Part IV Lustre for Users

24. Free Space and Quotas 24-1

24.1 Querying Filesystem Space 24-1

24.2 Using Quota 24-3

25. Striping and I/O Options 25-1

25.1 File Striping 25-1

25.1.1 Advantages of Striping 25-2

25.1.2 Disadvantages of Striping 25-3

25.1.3 Stripe Size 25-3

25.2 Displaying Files and Directories with lfs getstripe 25-4

25.3 lfs setstripe – Setting Striping Patterns 25-6

25.3.1 Changing Striping for a Subdirectory 25-7

25.3.2 Using a Specific Striping Pattern for a Single File 25-7

25.4 Free Space Management 25-8

25.4.1 Round-Robin Allocator 25-8

25.4.2 Weighted Allocator 25-9

25.4.3 Adjusting the Weighting Between Free Space and Location 25-9

25.5 Performing Direct I/O 25-10

25.5.1 Making Filesystem Objects Immutable 25-10

25.6 Other I/O Options 25-11

25.6.1 End-to-End Client Checksums 25-11

25.7 Striping Using llapi 25-12

26. Lustre Security 26-1

26.1 Using ACLs 26-1

26.1.1 How ACLs Work 26-1

26.1.2 Lustre ACLs 26-2

26.1.3 Examples 26-3

27. Lustre Operating Tips 27-1

27.1 Expanding the Filesystem by Adding OSTs 27-1

27.2 A Simple Data Migration Script 27-3

27.3 Adding Multiple SCSI LUNs on Single HBA 27-4

27.4 Failures While Running a Client and an OST on the Same Machine 27-5

27.5 Improving Lustre Metadata Performance While Using Large Directories 27-5

Part V Reference

28. User Utilities (man1) 28-1

28.1 lfs 28-1

28.1.1 Synopsis 28-1

28.1.2 Description 28-2

28.1.3 Examples 28-4

28.2 lfsck 28-8

28.2.1 Synopsis 28-8

28.2.2 Description 28-9

28.3 Mount 28-14

28.4 Handling Timeouts 28-15

29.	Lustre Programming Interfaces (man2)	29-1
29.1	User/Group Cache Upcall	29-1
29.1.1	Name	29-1
29.1.2	Description	29-2
29.1.3	Parameters	29-3
29.1.4	Data structures	29-3
30.	Setting Lustre Properties (man3)	30-1
30.1	Using llapi	30-1
30.1.1	llapi_file_create	30-1
30.1.2	llapi_file_get_stripe	30-4
30.1.3	llapi_file_open	30-5
31.	Configuration Files and Module Parameters (man5)	31-1
31.1	Introduction	31-1
31.2	Module Options	31-2
31.2.1	LNET Options	31-3
31.2.2	SOCKLND Kernel TCP/IP LND	31-8
31.2.3	QSW LND	31-10
31.2.4	RapidArray LND	31-11
31.2.5	VIB LND	31-12
31.2.6	OpenIB LND	31-13
31.2.7	Portals LND (Linux)	31-14
31.2.8	Portals LND (Catamount)	31-16
31.2.9	MX LND	31-18

32.	System Configuration Utilities (man8)	32-1
32.1	mkfs.lustre	32-1
32.1.1	Synopsis	32-1
32.1.2	Description	32-2
32.1.3	Examples	32-3
32.2	tunefs.lustre	32-3
32.2.1	Synopsis	32-3
32.2.2	Description	32-4
32.2.3	Examples	32-5
32.3	lctl	32-5
32.3.1	Synopsis	32-5
32.3.2	Description	32-5
32.3.3	Examples	32-11
32.3.4	Network Commands	32-12
32.4	mount.lustre	32-13
32.4.1	Synopsis	32-13
32.4.2	Description	32-13
32.4.3	Examples	32-14
32.5	New Utilities in Lustre 1.6	32-15
32.5.1	General Purpose Utilities	32-15
32.5.2	Utilities to Manage Large Clusters	32-15
32.5.3	Application Profiling Utilities	32-16
32.5.4	More /proc Statistics for Application Profiling	32-17
32.5.5	Testing / Debugging Utilities	32-18
32.5.6	Flock Feature	32-19

33. System Limits 33-1

- 33.1 Maximum Stripe Count 33-1
- 33.2 Maximum Stripe Size 33-2
- 33.3 Minimum Stripe Size 33-2
- 33.4 Maximum Number of OSTs and MDSs 33-2
- 33.5 Maximum Number of Clients 33-2
- 33.6 Maximum Size of a Filesystem 33-3
- 33.7 Maximum File Size 33-3
- 33.8 Maximum Number of Files or Subdirectories in a Single Directory 33-3
- 33.9 MDS Space Consumption 33-4
- 33.10 Maximum Length of a Filename and Pathname 33-4
- 33.11 Maximum Number of Open Files for Lustre Filesystems 33-4
- 33.12 OSS RAM Size for a Single OST 33-5

A. Feature List A-1

B. Task List B-1

C. Version Log C-1

D. Lustre Knowledge Base D-1

Glossary Glossary-1

Index Index-1

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	<i>machine-name%</i>
C shell superuser	<i>machine-name#</i>
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 <code>.login</code> file. Use <code>ls -a</code> to list all files. % You have mail.
AaBbCc123	What you type, when contrasted with on-screen computer output	% su Password:
<i>AaBbCc123</i>	Book titles, new words or terms, words to be emphasized. Replace command-line variables with real names or values.	Read Chapter 6 in the <i>User's Guide</i> . These are called <i>class</i> options. You <i>must</i> be superuser to do this. To delete a file, type <code>rm filename</code> .

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.

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	December 2007	

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.

A Cluster with Lustre

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

- [What is Lustre?](#)
- [Lustre Software](#)
- [Lustre Components](#)

1.1 What is Lustre?

Lustre is a high-performance, multi-network, fault-tolerant, POSIX¹-compliant network filesystem for Linux clusters.

The key features of Lustre:

- Capacity to run over a wide range of network fabrics
- Fine-grained locking for efficient concurrent file access
- Failover ability to reconstruct the state if a server node fails
- Distributed file object handling for scalable data access

Lustre is a complete, software-only, open-source solution for any hardware that can run Linux. It has native drivers for many of the fastest networking fabrics. Lustre can use any storage medium that looks like a block device.

1. Portable Operating System Interface for UNIX (POSIX)

1.2 Lustre Software

The Lustre software consists of three interactive areas:

- **Patched Linux kernel**

Lustre requires significant changes from the standard Linux kernel to facilitate some of its performance improvements. These changes are distributed in the form of patches against specific kernels. Several specific, pre-patched kernels are available at these Lustre download websites:

- **Lustre 1.6.0**

https://cds.sun.com/is-bin/INTERSHOP.enfinity/WFS/CDS-CDS_SMI-Site/en_US/-/USD/ViewProductDetail-Start?ProductRef=LUSTRE-160-G-F-B@CDS-CDS_SMI

- **Lustre 1.6.0.1**

https://cds.sun.com/is-bin/INTERSHOP.enfinity/WFS/CDS-CDS_SMI-Site/en_US/-/USD/ViewProductDetail-Start?ProductRef=LUSTRE-1601-G-F@CDS-CDS_SMI

- **Lustre 1.6.1**

https://cds.sun.com/is-bin/INTERSHOP.enfinity/WFS/CDS-CDS_SMI-Site/en_US/-/USD/ViewProductDetail-Start?ProductRef=LUSTRE-161-G-F@CDS-CDS_SMI

- **Lustre 1.6.2**

https://cds.sun.com/is-bin/INTERSHOP.enfinity/WFS/CDS-CDS_SMI-Site/en_US/-/USD/ViewProductDetail-Start?ProductRef=LUSTRE-162-G-F@CDS-CDS_SMI

- **Lustre 1.6.3**

https://cds.sun.com/is-bin/INTERSHOP.enfinity/WFS/CDS-CDS_SMI-Site/en_US/-/USD/ViewProductDetail-Start?ProductRef=LUSTRE-163-G-F@CDS-CDS_SMI

- **Lustre 1.6.4.1**

https://cds-edit.sun.com/is-bin/INTERSHOP.enfinity/WFS/CDS-CDS_SMI-Site/en_US/-/USD/ViewProductDetail-Start?ProductRef=LUSTRE-1641-G-F@CDS-CDS_SMI

Additionally, the Lustre client, but not Lustre servers, can run on certain unmodified kernels (known as "patchless" kernels).

- **Lustre modules**

Lustre's kernel modules provide server and client capabilities for the filesystem.

- Userspace utilities

Several userspace utilities are required for Lustre configuration and the startup and shutdown of Lustre servers and clients.

1.3 Lustre Components

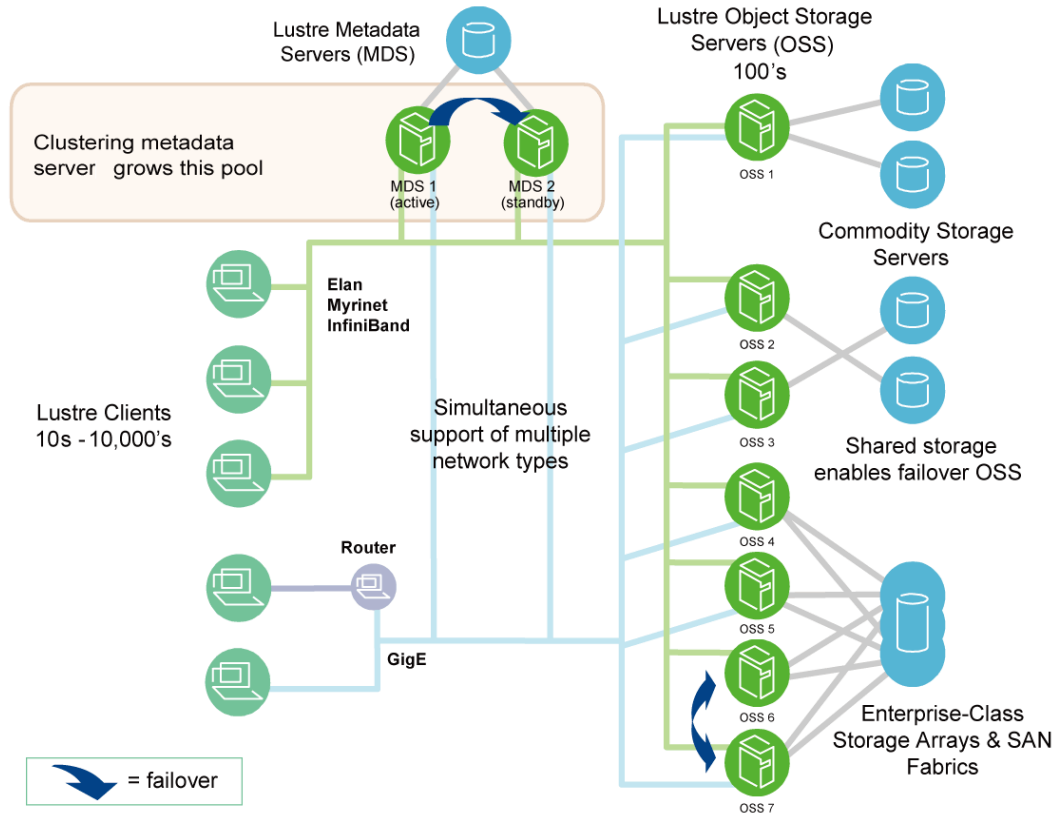
A Lustre filesystem consists of four major components:

- Management Server
- Metadata Target
- Object Storage Targets
- Lustre clients

Lustre clients provide remote access to a Lustre filesystem. The filesystem is served jointly by the Object Storage Targets (OSTs) for file contents and the Metadata Target (MDT) for file metadata (directory structure, file size, and so on).

A single Lustre filesystem may have multiple OSTs, each serving a subset of the file data. There is not necessarily a 1:1 correspondence between a file and an OST; a file may be spread over many OSTs to optimize performance. Each OST and the MDT may have a failover partner to provide access to the back-end storage if the server node fails. [FIGURE 1-1](#) shows the expected interactions between servers and clients in a Lustre filesystem.

FIGURE 1-1 Scaling with clustered metadata servers



The MDT, OSTs and Lustre clients can all run concurrently (in any mixture) on a single node. However, a more typical configuration is an MDT on a dedicated node, two or more OSTs on each Object Storage node, and a client on each of a large number of computer nodes.

1.3.1 MGS

The Management Server (MGS) defines configuration information for all Lustre filesystems at a site. Each Lustre target contacts the MGS to provide information, and Lustre clients contact the MGS to retrieve information. The MGS can provide live updates to the configuration of targets and clients. 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 filesystem; it provides configuration mechanisms to other Lustre components.

1.3.2 MDT

The MDT provides back-end storage for metadata for a single filesystem. The Metadata Server (MDS) provides the network request handling for one or more local MDTs.¹

The metadata managed by the MDT consists of the file hierarchy ("namespace"), along with file attributes such as permissions and references to the data objects stored on the OSTs.

1.3.3 OSTs

An OST provides back-end storage for file object data (effectively, chunks of user files). Typically, multiple OSTs provide access to different file chunks. The MDT tracks the location of the chunks. On a node serving OSTs, an Object Storage Server (OSS) component provides the network request handling for one or more local OSTs.

1.3.4 Lustre Client Nodes

Lustre clients are the "users" of the filesystem. Typically, the clients are computation, visualization, or desktop nodes. Lustre clients require Lustre software to mount a Lustre filesystem—Lustre is not NFS.

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 Logical Object Volume (LOV). Working in concert, the OSCs provide transparent access to the filesystem.

All clients which mount the filesystem 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. This is a common situation for large simulations and is an area in which Lustre excels.

Almost all activity on the Targets is driven by requests from Lustre clients.

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.

1.3.5 LNET

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).

LNET provides the delivery and event generation in connection with network messages. It also provides advanced capabilities such as using remote direct memory access (RDMA), if the underlying network transport layer supports it, and autonomous routing between different network transports on different nodes.

Understanding Lustre Networking

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

- [Introduction to LNET](#)
- [Supported Network Types](#)
- [Important Terms](#)

2.1 Introduction to LNET

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

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

2.2 Supported Network Types

Lustre supports the following network types:

- TCP (Ethernet)
- openib (Mellanox-Gold InfiniBand)
- iib (Infinicon InfiniBand)
- vib (Voltaire InfiniBand)
- o2ib (OFED)
- ra (RapidArray)
- Elan (Quadrics Elan)
- gm and mx (Myrinet)
- LNET

2.3 Important Terms

The following terms are important to understanding Lustre networking.

- **LND:** Lustre network driver. A modular sub-component of LNET that implements one of the network types. LNDs are implemented as individual Linux modules 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.

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.

Prerequisites

This chapter describes Lustre installation prerequisites, and includes the following sections:

- [Preparing to Install Lustre](#)
- [Using a Pre-Packaged Lustre Release](#)
- [Environmental Requirements](#)
- [Memory Requirements](#)

3.1 Preparing to Install Lustre

This chapter describes the prerequisites to install Lustre.

3.1.1 How to Get Lustre

The most recent versions of Lustre are available at these download websites:

- **Lustre 1.6.0**

https://cds.sun.com/is-bin/INTERSHOP.enfinity/WFS/CDS-CDS_SMI-Site/en_US/-/USD/ViewProductDetail-Start?ProductRef=LUSTRE-160-G-F-B@CDS-CDS_SMI

- **Lustre 1.6.0.1**

https://cds.sun.com/is-bin/INTERSHOP.enfinity/WFS/CDS-CDS_SMI-Site/en_US/-/USD/ViewProductDetail-Start?ProductRef=LUSTRE-1601-G-F@CDS-CDS_SMI

- **Lustre 1.6.1**

https://cds.sun.com/is-bin/INTERSHOP.enfinity/WFS/CDS-CDS_SMI-Site/en_US/-/USD/ViewProductDetail-Start?ProductRef=LUSTRE-161-G-F@CDS-CDS_SMI

- **Lustre 1.6.2**

https://cds.sun.com/is-bin/INTERSHOP.enfinity/WFS/CDS-CDS_SMI-Site/en_US/-/USD/ViewProductDetail-Start?ProductRef=LUSTRE-162-G-F@CDS-CDS_SMI

- **Lustre 1.6.3**

https://cds.sun.com/is-bin/INTERSHOP.enfinity/WFS/CDS-CDS_SMI-Site/en_US/-/USD/ViewProductDetail-Start?ProductRef=LUSTRE-163-G-F@CDS-CDS_SMI

- **Lustre 1.6.4.1**

https://cds-edit.sun.com/is-bin/INTERSHOP.enfinity/WFS/CDS-CDS_SMI-Site/en_US/-/USD/ViewProductDetail-Start?ProductRef=LUSTRE-1641-G-F@CDS-CDS_SMI

The Lustre software is released under the GNU General Public License (GPL). We strongly recommend that you read the complete GPL and release notes before downloading Lustre (if you have not done so already). The GPL and release notes can also be found at the aforementioned websites.

3.1.2 Supported Configurations

We support Lustre on the configurations listed in [TABLE 3-1](#).

TABLE 3-1 Supported Configurations

Aspect	Support Type
Operating Systems	Red Hat Enterprise Linux 3+ SuSE Linux Enterprise Server 9 and 10 Linux 2.4 and 2.6
Platforms	IA-32, IA-64, x86-64 PowerPC architectures and mixed-endian clusters
Interconnect	TCP/IP Quadrics Elan 3 and 4 Myri-10G and Myrinet - 2000 Mellanox InfiniBand (Voltaire, OpenIB and Silverstorm)

Different endians like, i386 and PPC, also support Lustre clients. One limitation is that the `PAGE_SIZE` 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 4kB `PAGE_SIZE`.

3.2 Using a Pre-Packaged Lustre Release

Due to the complexity involved in building and installing Lustre, we offer several pre-packaged releases that cover several of the most common configurations. A pre-packaged release consists of five different RPM packages (described below). Install these packages in the following order:

- `kernel-smp-<release-ver>.rpm` – This is the Lustre-patched Linux kernel RPM. Use it with matching Lustre Utilities and Lustre Modules packages.
- `kernel-source-<release-ver>.rpm` – This is the Lustre-patched Linux kernel source RPM. This RPM comes with the kernel package, but is not required to build or use Lustre.
- `lustre-modules-<release-ver>.rpm` – These are the Lustre kernel modules for the above kernel.
- `lustre-<release-ver>.rpm` – These are Lustre utilities or userspace utilities to configure and run Lustre. Only use them with the matching kernel RPM (referenced above).
- `lustre-source-<release-ver>.rpm` – This contains the Lustre source code (including the kernel patches). It is not required to build or use Lustre.

The source package is only required if you need to build your own modules (networking, for example) against the kernel source.

- `lustre-ldiskfs-<release-ver>.rpm` – This contains modules for the backend filesystem used by Lustre.



Caution – Lustre contains kernel modifications, which interact with your storage devices and may introduce security issues and data loss if not installed, configured, or administered properly. Before using this software, please exercise caution and back up ALL data.

3.2.1 Choosing a Pre-Packaged Kernel

Choosing the most suitable pre-packaged kernel depends largely on the combination of hardware and software being used where Lustre will be installed. Pre-packaged kernel releases are available at the Lustre download website.

3.2.2 Lustre Tools

The `lustre-<release-ver>.rpm` package, required for proper Lustre setup and monitoring, contains many tools. The most important tools are:

- `lctl` - Low-level configuration utility that can also be used for troubleshooting and debugging.
- `lfs` - Tool to read/set striping information for the cluster, as well as perform other actions specific to Lustre filesystems.
- `mount.lustre` - Lustre-specific helper for `mount(8)`.
- `mkfs.lustre` - Tool to format Lustre target disks.

3.2.3 Other Required Software

Although we provide some tools and utilities, Lustre also requires several separate software tools to be installed.

3.2.3.1 Core-Required Tools

- `e2fsprogs`: Lustre requires very modern `e2fsprogs` that understand extents—use `e2fsprogs-1.38-cfs1` or later, available at:

`ftp://ftp.lustre.org/pub/lustre/other/e2fsprogs/`

Note – You might have to install `e2fsprogs` with `rpm -ivh --force` to override any dependency issues of your distribution.

- `Perl`: Various userspace utilities are written in Perl. Any modern Perl should work with Lustre.
- `build tools`: If you are not installing Lustre from RPMs, normally you can use a GCC compiler to build Lustre. Use GCC 3.0 or later.

3.2.3.2 High-Availability Software

If you plan to enable failover server functionality with Lustre (either on an OSS or an MDS), high-availability software must be added to your cluster software. Heartbeat is one of the better known high-availability software packages.

Linux-HA (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](#).

3.2.3.3 Debugging Tools

Things inevitably go wrong—disks fail, packets get dropped, software has bugs, and when they do it is useful to have debugging tools on hand to help figure out how and why a problem occurred.

In this regard, the most useful tool is GDB, coupled with crash. You can use these tools 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 tools, see the following websites:

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/

3.3 Environmental Requirements

When preparing to install Lustre, make sure the following environmental requirements are met.

3.3.1 SSH Access

Although it is not strictly required, in many cases it is very helpful to have remote SSH¹ access to all the nodes in a cluster. Some Lustre configuration and monitoring scripts depend on SSH (or Pdsh²) access, although these are not required for running Lustre.

1. Secure SHell (SSH)
2. Parallel Distributed SHell (Pdsh)

3.3.2 Consistent Clocks

Lustre always uses the client clock for timestamps. If the machine clocks across the cluster are not in sync, Lustre should not break. However, the unsynchronized clocks in a cluster will always be a headache as it is very difficult to debug any multi-node issue, or otherwise correlate logs. For this reason, we recommend that you keep machine clocks in sync as much as possible. The standard way to accomplish this is by using the Network Time Protocol (NTP). All machines in your 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/>

3.3.3 Universal UID/GID

To maintain uniform file access permissions on all nodes in your cluster, use the same user IDs (UID) and group IDs (GID) on all clients. Like most cluster usage, Lustre uses a common UID/GID on all cluster nodes.

3.3.4 Choosing a Proper Kernel I/O Scheduler

One of the many functions of the Linux kernel (indeed of any OS kernel), is to provide access to disk storage. The algorithm which decides how the kernel provides disk access is known as the "I/O Scheduler," or "Elevator." In the 2.6 kernel series, there are four interchangeable schedulers:

Scheduler	Description
cfq	"Completely Fair Queuing" makes a good default for most workloads on general-purpose servers. It is not a good choice for Lustre OSS nodes, however, as it introduces overhead and I/O latency.
as	"Anticipatory Scheduler" is best for workstations and other systems with slow, single-spindle storage. It is not at all good for OSS nodes, as it attempts to aggregate or batch requests in order to improve performance for slow disks.
deadline	"Deadline" is a relatively simple scheduler which tries to minimize I/O latency by re-ordering requests to improve performance. Best for OSS nodes with "simple" storage, that is software RAID, JBOD, LVM, and so on.
noop	"NOOP" is the most simple scheduler of all, and is really just a single FIFO queue. It does not attempt to optimize I/O at all, and is best for OSS nodes that have high-performance storage, that is DDN, Engenio, and so on. This scheduler may yield the best I/O performance if the storage controller has been carefully tuned for the I/O patterns of Lustre.

The above observations on the schedulers are just our best advice. We strongly suggest that you conduct local testing to ensure high performance with Lustre. Also, note that most distributions ship with either “cfq” or “as” configured as the default scheduler. Choosing an alternate scheduler is an absolutely necessary step to optimally configure Lustre for the best performance. The “cfq” and “as” schedulers should never be used for server platform.

For more in-depth discussion on choosing an I/O scheduler algorithm for Linux, see:

- <http://www.redhat.com/magazine/008jun05/features/schedulers>
- http://www.novell.com/brainshare/europe/o5_presentations/tut303.pdf
- <http://kerneltrap.org/node/3851>

3.3.5 Changing the I/O Scheduler

There are two ways to change the I/O scheduler—at boot time or with new kernels at runtime. For all Linux kernels, appending `elevator={noop|deadline}` to the kernel boot string sets the I/O elevator.

With LILO, you can use the `append` keyword:

```
image=/boot/vmlinuz-2.6.14.2
label=14.2
append="elevator=deadline"
read-only
optional
```

With GRUB, append the string to the end of the kernel command:

```
title Fedora Core (2.6.9-5.0.3.EL_lustre.1.4.2custom)
root (hd0,0)
kernel /vmlinuz-2.6.9-5.0.3.EL_lustre.1.4.2custom ro
root=/dev/VolGroup00/LogVol00 rhgb noapic quiet elevator=deadline
```

With newer Linux kernels,³ one can change the scheduler while running. If the file `/sys/block/<DEVICE>/queue/scheduler` exists (where `<DEVICE>` is the block device you wish to affect), it contains a list of available schedulers and can be used to switch the schedulers.

(hda is the `<disk>`):

```
[root@cfs2]# cat /sys/block/hda/queue/scheduler
noop [anticipatory] deadline cfq
[root@cfs2 ~]# echo deadline > /sys/block/hda/queue/scheduler
[root@cfs2 ~]# cat /sys/block/hda/queue/scheduler
noop anticipatory [deadline] cfq
```

For desktop use, the other schedulers (anticipatory and cfq) are better suited.

3.4 Memory Requirements

This section describes the memory requirements of Lustre.

3.4.1 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 primarily driven by the number of locks a client can hold at one time. The default maximum number of locks for a compute node is $100 \times \text{num_cores}$, 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 filesystem 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.

3. Red Hat Enterprise Linux v3 Update 3 does not have this feature. It is present in the main Linux tree as of 2.6.15.

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):

filesystem 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.

3.4.2 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` filesystem has 400 MB for the journal size. This can pin up to an equal amount of RAM on the OSS node per filesystem. 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 filesystem metadata. While no hard limit can be placed on the amount of filesystem 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 filesystem 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.

Lustre Installation

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

- [Installing Lustre](#)
- [Quick Configuration of Lustre](#)
- [Building from Source](#)
- [Building a Lustre Source Tarball](#)

Currently, all Lustre installations run the ext3 filesystem internally on service nodes. Lustre servers run on top of the ext3 filesystem internally. The ext3 creates a journal for efficient recovery after a system crash or power outages. For maximum performance on a very large filesystem, Lustre creates a very large journal, up to 400 MB per target. If your filesystem runs on 100 OSTs, a total of 40 GB of space is used for the journals.

4.1 Installing Lustre

Use this procedure to install Lustre.

1. **Install the Linux base OS per your requirements and installation prerequisites like GCC and Perl, discussed in [Prerequisites](#).**
2. **Install the RPMs (described in [Using a Pre-Packaged Lustre Release](#)). The preferred installation order is:**
 - Lustre-patched version of the linux kernel (kernel-*)
 - Lustre-ldiskfs
 - Lustre modules for that kernel (lustre-modules-*)
 - Lustre userspace programs (lustre-*). Other packages (optional).
3. **Verify that all cluster networking is correct. This may include `/etc/hosts` or DNS. Set the correct networking options for Lustre in `/etc/modprobe.conf`. See [Modprobe.conf](#)**

Tip – When installing Lustre with InfiniBand, keep the ibhost, kernel and Lustre all on the same revision. To do this:

1. Install the kernel source (Lustre-patched)
2. Install the Lustre source and the ibhost source.
3. Compile the ibhost against your kernel.
4. Compile the Linux kernel.
5. Compile Lustre against the ibhost source `--with-vib=<path to ibhost>`.

Now you can use the RPMs created by the above steps.

4.1.1 MountConf

MountConf is shorthand for Mount Configuration. The Lustre cluster is configured only by the `mkfs.lustre` and `mount` commands. The MountConf system is one of the important features of Lustre 1.6.x.

MountConf involves userspace utilities (`mkfs.lustre`, `tunefs.lustre`, `mount.lustre`, `lctl`) and two new OBD types, the MGC and MGS. The MGS is a configuration management server, which compiles configuration information about all Lustre filesystems running at a site. There should be one MGS per site, not one MGS per filesystem. The MGS requires its own disk for storage. However, there is a provision to allow the MGS to share a disk ("co-locate") with an MDT of one filesystem.

You must start the MGS first as it manages the configurations. Beyond this, there are no ordering requirements to when a Target (MDT or OST) can be added to a filesystem. (However, there should be no client I/O at addition time, also known as "quiescent ost addition.")

For example, consider the following order of starting the servers.

1. **Start the MGS - start mgs**
2. **Mount OST 1 - mkfs, mount ost #1**
3. **Mount the MDT - mkfs, mount mdt**
4. **Mount OST 2 - mkfs, mount ost #2**
5. **Mount the client - mount client**
6. **Mount OST 3 - mkfs, mount ost #3**

The clients and the MDT are notified that there is a new OST on-line and they can use it immediately.

Note – The MGS must be running before any new servers are added to the filesystem. After the servers start the first time, they cache a local copy of their startup logs so they can restart with or without the MGS.

Currently, there is nothing actually visible on a server mount point (but `df` will show free space). Eventually, the mount point will probably look like a Lustre client.

4.2 Quick Configuration of Lustre

As already discussed, Lustre consists of four types of subsystems – a Management Server (MGS), a Metadata Target (MDT), Object Storage Targets (OSTs) and clients. All of these can co-exist on a single system or can run on different systems. Together the OSSs and MDS together 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 source code, the scripts are located in the `lustre/tests` sub-directory. These scripts enable quick setup of some simple, standard configurations.

The next section describes how to use these scripts to install a simple Lustre setup.

4.2.1 Simple Configurations

The procedures in this section describe how to set up simple Lustre configurations.

4.2.1.1 Module Setup

Make sure the modules (like LNET) are installed in the appropriate `/lib/modules` directory. The `mkfs.lustre` and `mount.lustre` utilities automatically load the correct modules.

1. **Set up module options for networking should first be set up by adding the following line in `/etc/modprobe.conf` –**

```
# Networking options, see /sys/module/lnet/parameters NO \
../lnet/parameters dir
```

2. **Add the following line.**

```
options lnet networks=tcp
# alias lustre llite -- remove this line from existing modprobe.conf
#(the llite module has been renamed to lustre)
# end Lustre modules
```

The clients and the MDT are notified that there is a new OST on-line and immediately are able to use it.

Note – For detailed information on formatting an MDS or OST, see [Options to Format MDT and OST Filesystems](#).

4.2.1.2 Making and Starting a Filesystem

Starting Lustre on MGS and MDT Node “mds16”

First, create an MDT for the "spfs" filesystem that uses the `/dev/sda` disk. This MDT also acts as the MGS for the site.

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

```
Permanent disk data:
```

```
Target:spfs-MDTffff
```

```
Index:unassigned
```

```
Lustre FS:spfs
```

```
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: not found
```

```
device size = 4096MB
```

```
formatting backing filesystem ldiskfs on /dev/sda
```

```
target name spfs-MDTffff
```

```
4k blocks0
```

```
options-J size=160 -i 4096 -I 512 -q -O dir_index -F
```

```
mkfs_cmd = mkfs.ext2 -j -b 4096 -L spfs-MDTffff -J \
```

```
size=160 -i 4096 -I 512 -q -O dir_index -F /dev/sda
```

```
Writing CONFIGS/mountdata
```

```
$ mkdir -p /mnt/test/mdt
```

```
$ mount -t lustre /dev/sda /mnt/test/mdt
```

```
$ cat /proc/fs/lustre/devices
```

```
0 UP mgs MGS MGS 5
```

```
1 UP mgc MGC192.168.16.21@tcp bf0619d6-57e9-865c-551c- \
06cc28f3806c 5
```

```
2 UP mdt MDS MDS_uuid 3
```

```
3 UP lov spfs-mdtlov spfs-mdtlov_UUID 4
```

```
4 UP mds spfs-MDT0000 spfs-MDT0000_UUID 3
```

Starting Lustre on any OST Node

Give OSTs the location of the MGS with the `--mgsnode` parameter.

```
$ mkfs.lustre --fsname spfs --ost --mgsnode=mds16@tcp0 /dev/sda
```

```
Permanent disk data:
Target:spfs-OSTffff
Index:unassigned
Lustre FS:spfs
Mount type:ldiskfs
Flags:0x72
(OST needs_index first_time update)
Persistent mount opts: errors=remount-ro,extents,malloc
Parameters: mgsnode=192.168.16.21@tcp
```

```
device size = 4096MB
formatting backing filesystem ldiskfs on /dev/sda
target namespace spfs-OSTffff
4k blocks0
options -J size=160 -i 16384 -I 256 -q -O dir_index -F
mkfs_cmd = mkfs.ext2 -j -b 4096 -L spfs-OSTffff -J \
size=160 -i 16384 -I 256 -q -O dir_index -F /dev/sda
Writing CONFIGS/mountdata
$ mkdir -p /mnt/test/ost0
$ mount -t lustre /dev/sda /mnt/test/ost0
$ cat /proc/fs/lustre/devices
0 UP mgc MGC192.168.16.21@tcp 7ed113fe-dd48-8518-a387- 5c34eec6fbf4 5
1 UP ost OSS OSS_uuid 3
2 UP obdfilter spfs-OST0000 spfs-OST0000_UUID 5
```

Mounting Lustre on a client node

```
$ mkdir -p /mnt/testfs
$ mount -t lustre cfs21@tcp:0:/testfs /mnt/testfs
```

The MGS and the MDT can be run on separate devices. With the MGS on node 'mgs16':

```
$ mkfs.lustre --mgs /dev/sda1
$ mkdir -p /mnt/mgs
$ mount -t lustre /dev/sda1 /mnt/mgs
$ mkfs.lustre --fsname=spfs --mdt --mgsnode=mgs16@tcp0 /dev/sda2
$ mkdir -p /mnt/test/mdt
$ mount -t lustre /dev/sda1 /mnt/test/mdt
```

If the MGS node has multiple interfaces (for example, mgs16 and 1@elan), only the client mount command has to change. The MGS NID specifier must be an appropriate nettype for the client (for instance, TCP client could use uml1@tcp0 and Elan client could use 1@elan). Alternatively, a list of all MGS NIDs can be provided and the client chooses the correct one.

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

Reformat a device that has already been formatted with `mkfs.lustre`

```
$ mkfs.lustre --fsname=spfs --mdt --mgs --reformat /dev/sda1
```

4.2.1.3 Filesystem Name

The filesystem name is limited to 8 characters. We have encoded the filesystem and target information in the disk label, so that 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, filesystem naming will be made as fail-safe as possible. Currently, Linux disk labels are limited to 16 characters. To identify the target within the filesystem, 8 characters are reserved, leaving 8 characters for the filesystem name:

```
myfsname-MDT0000 or myfsname-OST0a19
```

An example of mount-by-label:

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

Although the filesystem name is internally limited to 8 characters, you can mount the clients at any mount point, so filesystem users are not subjected to short names:

```
mount -t lustre uml1@tcp0:/shortfs /mnt/my-long-filesystem-name
```

4.2.1.4 Starting a Server Automatically

Starting Lustre only involves the `mount` command, Lustre servers can be added to `/etc/fstab`:

```
$ mount -l -t lustre
/dev/sda1 on /mnt/test/mdt type lustre (rw) [testfs-MDT0000]
/dev/sda2 on /mnt/test/ost0 type lustre (rw) [testfs-OST0000]
192.168.0.21@tcp:/testfs on /mnt/testfs type lustre (rw)
```

Add to `/etc/fstab`:

```
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, ensure that networking has been started before mounting a Lustre server. RedHat, SuSe, Debian (maybe 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.

4.2.1.5 Stopping a Server

To stop a server:

```
$ umount -f /mnt/test/ost0
```

The `-f` flag means “force”; force the server to stop **WITHOUT RECOVERY**. Without the `-f` flag, “failover” is implied, meaning the next time the server is started it goes through the recovery procedure.

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

4.2.2 More Complex Configurations

In case of NID/node specification (note that a node is a server box), it may have multiple NIDs if it has multiple network interfaces. When a node is specified, generally all of its NIDs are required to be listed, delimited by commas (,) so other nodes can choose the NID appropriate to their own network interfaces. When multiple nodes are specified, they are delimited by a colon (:) or by repeating a keyword (`--mgsnode=` or `--failnode=`). To obtain all the NIDs from a node (while LNET is running), run this command:

```
lctl list_nids
```

4.2.2.1 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.)

4.2.2.2 Mount with Inactive OSTs

Mounting a client or MDT with known down OSTs (specified targets are treated as "inactive")

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

To reactivate an inactive OST on a live client or MDT, use `lctl activate` 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`.

4.2.2.3 Without Lustre Service

Only start the MGS or MGC. Do not start the target server (for example, if you do not want to start the MDT for a combined MGS/MDT)

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

4.2.2.4 Failout

Designate an OST as a "failout", so clients receive errors after a timeout instead of waiting for recovery:

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


4.2.2.5 Running Multiple Lustres

The default filesystem name created by `mkfs.lustre` is `lustre`. For a different filesystem name, specify `mkfs.lustre --fsname=foo`. The MDT, OSTs and clients that comprise a single filesystem must share the same name. For example:

```
foo-MDT0000
```

```
foo-OST0000
```

```
foo-OST0001
```

```
client mount command: mount -t lustre mgsnode:/foo /mnt/mountpoint
```

The maximum length of the filesystem name is 8 characters.

The MGS is universal; there is only one MGS per installation, not per filesystem. An installation with two filesystems could look like this:

```
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
```

Client mount for foo:

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

Client mount for bar:

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

4.2.3 Other Configuration Tasks

This section describes other Lustre configuration tasks.

4.2.3.1 Removing an OST

In Lustre 1.6, an OST can be permanently removed from a filesystem. Any files that have stripes on the removed OST will, in the future, return EIO.

```
$ mgs> lctl --device {device number}conf_param testfs-OST0001.osc.active=0
```

This tells any clients of the OST that it should not be contacted; the OSTs current state is irrelevant.

To remove an OST:

1. **Deactivate the OST (make it read-only) on the MDS so no new objects are allocated to it.**
2. **Use `lfs find` to discover all files that have objects residing on the deactivated OST.**
3. **Copy these files to a new location and then move them back to their original location to force their object re-creation on the active OSTs and the object deletion on the OST to be removed.**

To restore the OST:

1. **Make sure the OST is running.**
2. **Run this command:**

```
$ mgs> lctl --device {device number}conf_param testfs-OST0001.osc.active=1
```

4.2.3.2 Running the Writeconf Command

To run `writeconf`, first remove all existing configuration files for a filesystem. Use the `writeconf` command on an MDT to erase all the configuration logs for the filesystem. The logs are regenerated only as servers restart; therefore all servers must be restarted before clients can access filesystem data. The logs are regenerated as in a new filesystem; old settings from `lctl conf_param` are lost, and current server NIDs are used. Only use the `writeconf` command if:

- The configuration logs are into a state where the filesystem cannot start; or
- You are changing the NIDs of one of the servers.

To run the `writeconf` command:

1. **Unmount all the clients and servers.**
2. **With every server disk, run:**

```
$ mdt> tuneufs.lustre --writeconf /dev/sda1
```

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

4.2.3.3 Changing a Server NID

To change a server NID:

1. **Update the LNET configuration in `/etc/modprobe.conf` so the `lctl list_nids` is correct.**
2. **Regenerate the configuration logs for every affected filesystem using the `--writeconf` flag to `tuneufs.lustre`, as shown in the second step of the section.**
3. **If the MGS NID is also changing, communicate the new MGS location to each server. Type:**

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

4.2.3.4 Aborting Recovery

When starting a target, abort the recovery process. Type:

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

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

4.3 Building from Source

This section describes how to build Lustre from source code.

4.3.1 Building Your Own Kernel

If you are using non-standard hardware or Lustre Support has asked you to apply a patch, you need to build your own kernel. Lustre requires some changes to the core Linux kernel. These changes are organized in a set of patches in the `kernel_patches` directory of the Lustre repository. If you are building your kernel from the source code, then you need to apply the appropriate patches.

Managing patches for the kernels is a very involved process, because most patches are intended to work with several kernels. We recommend that you use the Quilt package developed by Andreas Gruenbacher, as it simplifies the process considerably. Patch management with Quilt works as follows:

- 1. A series file lists a collection of patches.**
- 2. The patches in a series form a stack**
- 3. Using Quilt, you push and pop the patches.**
- 4. You then edit and refresh (update) the patches in the stack that is being managed with Quilt.**
- 5. You can then revert inadvertent changes and fork or clone the patches and conveniently show the difference in work (before and after).**

4.3.1.1 Selecting a Patch Series

Depending on the kernel being used, a different series of patches needs to be applied. A collection of different patch series files are maintained for the various supported kernels in this directory: `lustre/kernel_patches/series/`.¹

For example, the `lustre/kernel_patches/series/rh-2.4.20` file lists all patches that should be applied to the Red Hat 2.4.20 kernel to build a Lustre-compatible kernel.

The current set of all the supported kernels and their corresponding patch series can be found in the `lustre/kernel_patches/which_patch` file.

1. This directory is in the Lustre tarball.

4.3.1.2 Installing Quilt

A variety of Quilt packages (RPMs, SRPMs and tarballs) are available from various sources. We recommend that you use a recent version of Quilt. If possible, use a Quilt package from your distribution vendor.

If you cannot find an appropriate Quilt package or fulfill its dependencies, we suggest that you build Quilt from the tarball. You can download the tarball from the main Quilt website:

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

4.3.1.3 Preparing the Kernel Tree Using Quilt

To prepare the kernel tree to use Quilt:

- 1. After acquiring the Lustre source (CVS or tarball) and choosing a series file to match your kernel sources, choose a kernel config file.**

The `lustre/kernel_patches/kernel_configs` folder contains supported `.config` files, which are named to indicate which kernel and architecture with which they are associated. For example, the configuration file for the 2.6.9 kernel shipped with RHEL 4 (suitable for x86_64 SMP systems) is:

`kernel-2.6.9-2.6-rhel4-x86_64-smp.config`

- 2. Unpack the appropriate kernel source tree.**

This manual assumes that the resulting source tree (referred to as the destination tree) is in `/tmp/kernels/linux-2.6.9`

You are ready to use Quilt to manage the patching process for your kernel.

- 3. Perform the following commands to set up the necessary symlinks between the Lustre kernel patches and your kernel sources (assuming the Lustre sources are unpacked under `/tmp/lustre-1.4.7.3` and you have chosen the 2.6-rhel4 series):**

```
$ cd /tmp/kernels/linux-2.6.9
$ rm -f patches series
$ ln -s /tmp/lustre-1.5.97/lustre/kernel_patches/series/2.6-\
rhel4.series ./series
$ ln -s /tmp/lustre-1.5.97/lustre/kernel_patches/patches .
```

- 4. You can now use Quilt to apply all patches in the chosen series to your kernel sources. Run:**

```
$ cd /tmp/kernels/linux-2.6.9
$ quilt push -av
```

If the right series files are chosen, and the patches and the kernel sources are up-to-date, the patched destination Linux tree should be able to act as a base Linux source tree for Lustre.

You do not need to compile the patched Linux source in order to build Lustre from it. However, you must compile the same Lustre-patched kernel and then boot it on any node on which you intend to run the version of Lustre being built using this patched kernel source.

4.3.2 Building Lustre

The most recent versions of Lustre are available for download:

Lustre 1.6.0

https://cds.sun.com/is-bin/INTERSHOP.enfinity/WFS/CDS-CDS_SMI-Site/en_US/-/USD/ViewProductDetail-Start?ProductRef=LUSTRE-160-G-F-B@CDS-CDS_SMI

Lustre 1.6.0.1

https://cds.sun.com/is-bin/INTERSHOP.enfinity/WFS/CDS-CDS_SMI-Site/en_US/-/USD/ViewProductDetail-Start?ProductRef=LUSTRE-1601-G-F@CDS-CDS_SMI

Lustre 1.6.1

https://cds.sun.com/is-bin/INTERSHOP.enfinity/WFS/CDS-CDS_SMI-Site/en_US/-/USD/ViewProductDetail-Start?ProductRef=LUSTRE-161-G-F@CDS-CDS_SMI

Lustre 1.6.2

https://cds.sun.com/is-bin/INTERSHOP.enfinity/WFS/CDS-CDS_SMI-Site/en_US/-/USD/ViewProductDetail-Start?ProductRef=LUSTRE-162-G-F@CDS-CDS_SMI

Lustre 1.6.3

https://cds.sun.com/is-bin/INTERSHOP.enfinity/WFS/CDS-CDS_SMI-Site/en_US/-/USD/ViewProductDetail-Start?ProductRef=LUSTRE-163-G-F@CDS-CDS_SMI

Lustre 1.6.4.1

https://cds.sun.com/is-bin/INTERSHOP.enfinity/WFS/CDS-CDS_SMI-Site/en_US/-/USD/ViewProductDetail-Start?ProductRef=LUSTRE-1641-G-F@CDS-CDS_SMI

Lustre 1.6.4.2

https://cds.sun.com/is-bin/INTERSHOP.enfinity/WFS/CDS-CDS_SMI-Site/en_US/-/USD/ViewProductDetail-Start?ProductRef=LUSTRE-1642-G-F@CDS-CDS_SMI

The following set of packages are available for each supported Linux distribution and architecture. The files use the following naming convention:

`kernel-smp-<kernel version>_lustre.<lustre version>.<arch>.rpm`

This is an example of binary packages for version 1.5.97:

- `kernel-lustre-smp-2.6.9-42.0.3.EL_lustre.1.5.97.i686.rpm` contains the patched kernel
- `lustre-1.5.97-2.6.9_42.0.3.EL_lustre.1.5.97smp.i686.rpm` contains Lustre userspace files and utilities
- `lustre-modules-1.5.97-2.6.9_42.0.3.EL_lustre.1.5.97smp.i686.rpm` contains Lustre modules (kernel/fs/lustre and kernel/net/lustre)

Use standard RPM commands to install the binary packages:

```
$ rpm -ivh kernel-lustre-smp-2.6.9-42.0.3.EL_lustre.1.5.97.i686.rpm
$ rpm -ivh lustre-1.5.97-2.6.9_42.0.3.EL_lustre.1.5.97smp.i686.rpm
$ rpm -ivh lustre-modules-1.5.97-2.6.9_42.0.3.EL_lustre.1.5.97smp.i686.rpm
```

This is an example of Source packages:

- `kernel-lustre-source-2.6.9-42.0.3.EL_lustre.1.5.97.i686.rpm` contains source for the patched kernel
- `lustre-source-1.5.97-2.6.9_42.0.3.EL_lustre.1.5.97smp.i686.rpm` contains source for Lustre modules and userspace utilities

Note – Kernel-source and Lustre-source packages are provided in case you need to build external kernel modules or use additional network types. They are not required to run Lustre.

Once you have your Lustre source tree, run these commands to build Lustre.

```
$ cd <path to kernel tree>
$ cp /boot/config-'uname -r' .config
$ make oldconfig || make menuconfig
```

■ For 2.6 kernels, run:

```
$ make include/asm
$ make include/linux/version.h
$ make SUBDIRS=scripts
$ make include/linux/utsrelease.h
```

■ For 2.4 kernels, run:

```
$ make dep
```

To configure Lustre and to build Lustre RPMs, go to the Lustre source directory and run:

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

This creates a set of .rpms in /usr/src/redhat/RPMS/<arch> with a date-stamp appended (the SUSE path is /usr/src/packages).

Example:

```
lustre-1.5.97-\
2.6.9_42.xx.xx.EL_lustre.1.5.97.custom_200609072009.i686.rpm
lustre-debuginfo-1.5.97-\
2.6.9_42.xx.xx.EL_lustre.1.5.97.custom_200609072009.i686.rpm
lustre-modules-1.5.97-\
2.6.9_42.xx.xx.EL_lustre.1.5.97.custom_200609072009.i686.rpm
lustre-source-1.5.97-\
2.6.9_42.xx.xx.EL_lustre.1.5.97.custom_200609072009.i686.rpm
```

Change directory (cd) into the kernel source directory and run:

```
$ make rpm
```

This creates a kernel RPM suitable for the installation.

Example:

```
kernel-2.6.95.0.3.EL_lustre.1.5.97custom-1.i386.rpm
```


4.3.2.1 Configuration Options

Lustre supports several different 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. A complete list of supported features and packages can be obtained by issuing the command `./configure --help` in the Lustre source directory. The config files matching the kernel version are in the `configs/` directory of the kernel source. Copy one to `.config` at the root of the kernel tree.

4.3.2.2 Liblustre

The Lustre library client, `liblustre`, relies on `libsysio`, which is a library that provides POSIX-like file and name space support for remote filesystems from the application program address space. `Libsysio` can be obtained at the SourceForge website:

<http://sourceforge.net/projects/libsysio/>



Caution – Remember that `liblustre` is not for general use. It was created to work with specific hardware (Cray) and should NEVER be used with other hardware.

Development of `libsysio` has continued ever since it was first targeted for use with Lustre. First, check out the `b_lustre` branch from the `libsysio` repository. This gives the version of `libsysio` compatible with Lustre.

To build `libsysio`, run:

```
$ sh autogen.sh
$ ./configure --with-sockets
$ make
```

■ To build `liblustre`, run:

```
$ ./configure --with-lib -with-sysio=/path/to/libsysio/source
$ make
```

Compiler Choice

The compiler must be greater than GCC version 3.3.4. Currently, GCC v4.0 is not supported. GCC v3.3.4 has been used to successfully compile available pre-packaged releases, and it is the only officially-supported compiler. You may have mixed results with other compilers or even with other GCC versions.

Note – GCC v3.3.4 was used to build 2.6 series kernels.

4.3.3 Building from Source

Currently, the distributed kernels do not include third-party InfiniBand modules. Lustre packages cannot include IB network drivers for Lustre, however, Lustre does distribute the source code. Build your InfiniBand software stack against the provided kernel, and then build new Lustre packages. This includes following procedures.

InfiniBand

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

To build Silverstorm with Lustre, configure Lustre with:

```
--iib=<path to silverstorm sources>
```

OpenIB 1.0

Note – Currently (v1.4.5), the Voltaire IB module (kvibnal) does not work on the Altix system. This is due to hardware differences in the Altix system.

4.4 Building a Lustre Source Tarball

This section describes how to build tarballs from RPMs.

4.4.1 Lustre Source Tarball from Lustre Source RPM

To build a proper Lustre source tarball from the Lustre source RPM:

1. **Install the RPM.**
2. **Configure the resulting Lustre tree.**
3. **Run 'make dist'**

This produces a proper Lustre tarball. Untar it and name the resulting directory:
`lustre-<extraversion>`.

The `lbuild` script requires a working directory. This directory must be empty prior to starting `lbuild`. If the build fails, clean out the working directory before attempting to restart.

The following example shows a local build with no downloading. The name of the 'target' (kernel version) must match a file in `lustre/kernel_patches/targets`. If you do not specify `--target-arch` (the hardware platform), then all architectures will be built.

This example is for the RHEL 2.6 kernel.

```
$ lustre/build/lbuild--extraversion=1.4.9 \  
    --target=2.6-rhel4 \  
    --target-archs=i686 \  
    --release \  
    --kerneldir=/path_to_tarball/ \  
    --stage=/path_to_working_dir/ \  
    --lustre=/path_to_lustre_tarball/ \  
    --nodownload
```

4.4.2 Lustre Source Tarball from CVS

The following example shows how to build a tarball from CVS², and includes the building of additional network drivers (gm and vib).

You must properly configure the network driver tree prior to starting `lbuild`. The network drivers are compiled as part of the Lustre build. Options after the '--' separator are passed directly to the Lustre `./configure` script.

In the example:

- Replace `CVSROOT` with the proper CVS string
- `--tag` is the CVS tag for the version you are building

```
$ lustre/build/lbuild--extraversion=1.4.7.1 \  
    --target=2.6-rhel4 \  
    --target-archs=i686 \  
    --release \  
    --kerneldir=/path_to_tarball/ \  
    --stage=/path_to_working_dir/ \  
    --d:ext:$CVSROOT \  
    --tag=b_release_1_4_7 \  
    --disable-datestamp \  
    -- /* following options will be passed to lustre */ \  
    --with-gm=/path_to_gm/gm-2.1.23_Linux \  
    --with-vib=/path_to_vib/ibhost-3.5.0_13
```

2. CVS is not generally available. If you need CVS access or additional information, contact CFS.

Configuring the Lustre Network

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

- [Designing Your Lustre Network](#)
- [Configuring Your Lustre Network](#)
- [Starting and Stopping LNET](#)

5.1 Designing Your Lustre Network

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

5.1.1 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.

5.1.2 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).

5.1.3 Identify Network Interfaces to Include/Exclude from LNET

By default, LNET uses all interfaces for a given network type. If there are interfaces it should not use, (such as administrative networks, IP over IB, and so on), then the included interfaces should be explicitly listed.

5.1.4 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 help ease the maintenance of large clusters, it is possible to configure the networking setup for all nodes through 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](#).

5.1.5 Determine Appropriate Mount Parameters for Clients

In their 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.

5.2 Configuring Your Lustre Network

This section describes how to configure your Lustre network.

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.

5.2.1 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 this node should use all available TCP and Elan interfaces.

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. For 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.

- 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.
- Under Linux 2.4, `sysfs` is not available, but the LND-specific parameters are accessible via equivalent paths under `/proc`.

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.

5.2.1.1 SilverStorm InfiniBand Options

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

```
options lnet networks="iib3(2) "
```

This says that this node is on iib network number 3, using HCA[2] == ib3.

5.2.1.2 Setting Up the Default Debug Level

If you are using zeroconf (mount -t lustre), add a line to your modules.conf as follows:

```
post-install portals sysctl -w portals.debug=0x3f0400
```

This sets the debug level to the value you specify, whenever the portals module is loaded.

Note – The above value is the default value in Lustre. It provides useful information for diagnosing problems without materially impairing performance.

5.2.2 Module Parameters - Routing

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

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

Examples:

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

This identifies the Elan NIDs 2@elan0 and 10@elan0 as routers for the TCP network.

This is a more complicated example:

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

This specifies bi-directional routing; Elan clients can reach Lustre resources on the TCP networks and TCP clients can access the Elan networks. For more information on ip2nets, see [Modprobe.conf](http://modprobe.conf).

Here is a very complex, routed configuration with Voltaire Infiniband and Myrinet (GM) systems, with four systems configured as follows:

```
options lnet\
    ip2nets="gm10.0.3.*# aa*-io;\
    vib10.10.131.[11-18]-ipoib0;\
    vib10.10.132.*# cc*-ipoib;"\
    routes="gm10.10.131.[11-18]@vib# vib->gm via aa[11-13];\
    vib0xdd7f813b@gm# gm->vib via aa11;\
    vib0xdd7f81c7@gm# gm->vib via aa12;\
    vib0xdd7f81c2@gm# gm->vib via aa13"
```

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. Currently, only the clients using the sock LND and Elan LND avoid failed routers. We are working on extending this behavior to include all types of LNDs. The `auto_down` parameter enables/disables (1/0) the automatic marking of router state.

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

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

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

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

The router checker gets the following variables for each router:

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

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

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

5.2.2.1 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 be enough to handle the required file serving bandwidth (and a 25% margin for headroom).

5.2.3 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.

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.

5.3 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.

5.3.1 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.

5.3.1.1 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
```

5.3.2 Stopping LNET

Before the LNET modules can be removed, LNET references must be removed. In general, these references are removed automatically during Lustre shutdown, but for standalone routers, an explicit step is necessary to stop LNET. Run this command:

```
lctl network unconfigure
```

Note – Attempting to remove the 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). Be sure that the Lustre network and Lustre are stopped prior to unloading the modules. Be extremely careful using `rmmod -f`

To unconfigure the lctl network, run:

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

Tip – To remove all Lustre modules, run:

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

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

6.1.1.1 Installation Summary

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

6.1.1.2 Configuration Generation and Application

1. **Install the Lustre RPMS (per [Installing Lustre](#)) on all nodes that are going to be a part of the Lustre filesystem. 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 filesystem 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@tcp1 /dev/sdd
mkfs.lustre --fsname datafs --ost --mgsnode=mds16@tcp2 /dev/sda1
mkfs.lustre --fsname datafs --ost --mgsnode=mds16@tcp3 /dev/sdb
```

6. **Make a mount point on all the OSTs for the filesystem 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 filesystem “datafs” having an MGS and an MDT on separate nodes, four OSTs, and a number of Lustre clients.

6.1.2.1 Installation Summary

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

6.1.2.2 Configuration Generation and Application

1. **Install the Lustre RPMs (per [Installing Lustre](#)) on all the nodes that are going to be a part of the Lustre filesystem. 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 filesystem 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 filesystem 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@tcp1 /dev/sdd
mkfs.lustre --fsname datafs --ost --mgsnode=mds16@tcp2 /dev/sda1
mkfs.lustre --fsname datafs --ost --mgsnode=mds16@tcp3 /dev/sdb
```

8. Make a mount point on all the OSTs for the filesystem 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	Description
<i>hostname</i>	Hostname of the node in the cluster
<i>MD</i>	Marker of the MD device line
<i>md name</i>	MD device name, for example: /dev/md0
<i>operation mode</i>	Operations mode, either create or remove . Default is create .
<i>options</i>	A “catchall” for other mdadm options, for example: “-c 128”
<i>raid level</i>	RAID level: 0,1,4,5,6,10, linear and multipath
<i>hostname</i>	Hostname of the node in the cluster.
<i>component devices</i>	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	Description
<i>hostname</i>	Hostname of the node in the cluster
<i>PV</i>	Marker of the PV line
<i>pv names</i>	Devices or loopback files to be initialized for later use by LVM or to wipe the label, for example: <code>/dev/sda</code> Multiple devices or files are separated by space or by using shell expansions, for example: <code>"/dev/sd{a,b,c}"</code>
<i>operation mode</i>	Operations mode, either create or remove . Default is create .
<i>options</i>	A "catchall" for other pvcreate/pvremove options, for example: <code>"-vv"</code>

Linux LVM VG (Volume Group)

The CSV line format is:

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

Where:

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

Linux LVM LV (Logical Volume)

The CSV line format is:

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

Where:

Variable	Description
<i>hostname</i>	Hostname of the node in the cluster
<i>LV</i>	Marker of the LV line
<i>lv name</i>	Name of the logical volume to be created (optional) or path of the logical volume to be removed (required by the remove mode)
<i>operation mode</i>	Operations mode, either create or remove . Default is create .
<i>options</i>	A “catchall” for other lvcreate/lvremove options, for example: “-i 2 -l 128”
<i>lv size</i>	Size [kKmMgGtT] to be allocated for the new LV. Default is megabytes (MB).
<i>vg name</i>	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	Description
<i>hostname</i>	Hostname of the node in the cluster. It must match “uname -n”
<i>module_opts</i>	Lustre networking module options. Use the newline character (\n) to delimit multiple options.
<i>device name</i>	Lustre target (block device or loopback file)
<i>mount point</i>	Lustre target mount point
<i>device type</i>	Lustre target type (mgs, mdt, ost, mgs mdt, mdt mgs)
<i>fsname</i>	Lustre filesystem name (limit is 8 characters)
<i>mgs nids</i>	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)
<i>index</i>	Lustre target index
<i>format options</i>	A “catchall” contains options to be passed to mkfs.lustre. For example: “device-size”, “--param”, and so on.
<i>mkfs options</i>	Format options to be wrapped with --mkfsoptions=”” and passed to mkfs.lustre
<i>mount options</i>	If this script is invoked with “-m” option, then the value of this item is wrapped with --mountsoptions=”” and passed to mkfs.lustre; otherwise, the value is added into /etc/ fstab
<i>failover nids</i>	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 (‘:’).

Note – 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 filesystem 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 filesystem 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 – You can 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
```

More Complicated Configurations

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

- [Multihomed Servers](#)
- [Elan to TCP Routing](#)

7.1 Multihomed Servers

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
```

IB-only clients

```
options lnet networks="iib0"  
options kiiblnlnd ipif_basename=ib0
```

Note – In the case of TCP-only clients, all available IP interfaces are used for tcp0 since the interfaces are not specified. If there is more than one, the IP of the first one found is used to construct the tcp0 ID.

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 correct 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 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

Lusre users have options available on the following networks.

- On a dual HCA OSS server

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

- On the client with the odd IP address

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

- On the 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 TCP client, run:

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

Failover

This chapter describes failover in a Lustre system and includes the following sections:

- [What is Failover?](#)
- [OST Failover Review](#)
- [MDS Failover Review](#)
- [Configuring MDS and OSTs for Failover](#)
- [Setting Up Failover with Heartbeat V1](#)
- [Using MMP](#)
- [Setting Up Failover with Heartbeat V2](#)
- [Considerations with Failover Software and Solutions](#)

8.1 What is Failover?

We say a computer system is Highly Available when the services it provides are available with minimum 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.

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.

The Lustre filesystem 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. We provide the necessary scripts to interact with these packages, and exposes health information for system monitoring. The recommended choice is the Heartbeat package (from www.linux-ha.org). Heartbeat is responsible for detecting failure of the primary server node and controlling the failover. 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.

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 absolutely vital that no two active nodes access the same partition, 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. HA software controls the Lustre resources with a service script. For this purpose, we provide `/etc/init.d/lustre`.

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.

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

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. While sending a new request this type of connection, the connection does not develop until either the reply arrives or the connection becomes disconnected or failed. If there is no traffic on a given connection, you should check the connection periodically to ensure its status.

If an active connection disconnects, then it leads to at least one timed-out request. New and old requests alike are in sleep until:

- The reply arrives (in case of reactivation 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 -EIO for these requests) or the connection becomes "failed."

Therefore, the timeout is effectively infinite. Lustre waits as long as it needs to and avoids giving the application an -EIO error.

Finally, if a connection goes to "failed" condition, which happens immediately in the "failout" OST mode, new and old requests receive -EIO immediately. 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.

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 of the 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 with respect to the shared disk, the shared disk needs to provide multiple partitions, and 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 Review

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 Review

The MDS has only one failover mode: active/passive, as only one MDS may be active at a given time.

8.4 Configuring MDS and OSTs for Failover

8.4.1 Starting/Stopping a Resource

You can start a resource with the `mount` command and stop it with the `umount` command. For details, see [Stopping a Server](#).

8.4.2 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.

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.

8.4.3 Hardware Requirements for Failover

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

8.4.3.1 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).
- 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”. The 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.

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](#)).

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.

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/ttySO
```

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)
Aug 9 09:50:48 oss161 crmd: [4733]: notice: populate_cib_nodes:
Node: oss161.clusterfs.com (uuid: e370be9a-24f4-46a5-99ac-
41a88c5fa344)
Sep 7 10:42:40 dl_q_0 heartbeat: info: Running \
/etc/ha.d/resource.d/ostl 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 really 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
```


8.5.2 Mon (Status Monitor)

Mon requires two scripts:

- Monitor script (checks a resource for health)
- Alert script (triggered by failure of the monitor)

Mon requires one configuration file:

```
/etc/mon/mon.cf
```

We use a trap-based monitor. The trap is set with a time interval. The trap is cleared by checking Lustre health. If the trap is not cleared, mon triggers a failover.

All monitors are configured in one file. Mon is started as a service at boot prior to heartbeat startup. All monitors are disabled at startup and enabled by Heartbeat in conjunction with resource startup/shutdown.

8.5.2.1 Mon Setup and Configuration

This section describes installation prerequisites for Mon and how to install Mon.

Install Prerequisites for Mon

Mon is not required for a basic failover setup. It is not required for Heartbeat V2, as monitoring is included in V2.

The Heartbeat software monitors the health of the node. Adding Mon to the setup allows application health to be monitored (Lustre is the application).

The base package is available from:

```
ftp://ftp.kernel.org/pub/software/admin/
```

Mon requires following Perl packages:

- Time::Period
- Time::HiRes
- Convert::BER
- Mon::SNMP

As always, we recommend that you use CPAN when installing Perl. The packages are also available as tarballs refer to:

```
http://www.cpan.org
```

Install Mon

After installing the Perl packages, obtain the Mon tarball at:

```
ftp://ftp.kernel.org/pub/software/admin/mon/
```

- 1. Untar the tarball.**
- 2. Copy the Mon program to a location on the root path.**
(`/usr/lib/mon/mon` is the default)
- 3. Install the *moncmd* program.**
- 4. For this setup, we have altered the Mon startup a bit. You must patch the *S99mon* script, and install the result as `/etc/init.d/mon` – set this routine to start at boot, prior to the Heartbeat startup**

```
$ chkconfig --add mon
```

- 5. Verify that the path for *moncmd* in the init script matches where you installed *moncmd* (`/usr/local/bin/moncmd` is the default).**
- 6. Create a set of Mon directories as specified in `/etc/mon/mon.cf`**

```
cfbasedir = /etc/mon
alertdir = /usr/local/lib/mon/alert.d
mondir = /usr/local/lib/mon/mon.d
statedir = /usr/local/lib/mon/state.d
logdir = /usr/local/lib/mon/log.d
dtlogfile = /usr/local/lib/mon/log.d/downtime.log
```

- 7. Create the `/etc/mon/auth.cf` file - allow everything in the *command* section change `AUTH_ANY` to *all*.**
 - 8. Create the `/etc/mon/mon.cf` file. Starting with the provided example:**
 - a. Verify that the correct paths are set.**
 - b. For each Lustre object, create two watches:**
 - The first watch runs the trap monitor.
 - The second watch receives the trap.
 - Both monitors will attempt to fail Lustre if they fail.
 - The monitor currently hard kills heartbeat to guarantee failover
- A Lustre user has provided a shell script that generates a `mon.cf` file.

9. **Copy the supplied trap generator script (mon.trap) to a proper location (/usr/local/lib/mon/)**
This Perl script is based on a script found on the Mon mailing list. Other scripts are also available there.
10. **Copy the provided Lustre monitor script (lustre.mon.trap) to the mon monitor directory (/usr/local/lib/mon/mon.d)**
 - a. **Verify that the location of TRAPPER points at the trap generation script from mon.trap.**
 - b. **Verify that the name matches the script specified in /etc/mon/mon.cf**
This script is based on /etc/init.d/lustre
11. **Copy the provided Lustre alert script to the mon alert directory: /usr/local/lib/mon/alert.d**
 - a. **Verify that the name matches script specified in /etc/mon/mon.cf**
This is a stock script from the mon package.
 - b. **For the Lustre failover sequence, you are free to choose another method of triggering the transition.**
 - The script will not STONITH the node.
 - Edit the script to provide hard node power off or reboot (if needed).

Add Mon to the Heartbeat Configuration

1. **Copy the lustre-resource-monitor script to the Heartbeat resource directory (/etc/ha.d/resource.d)**
2. **Give the script a unique name (alpha-mon, beta-mon).**
3. **Edit the script, and set MONLIST to the service names to be monitored (two services per object as defined in /etc/mon/mon.cf).**
4. **Edit /etc/ha.d/haresources to add the mon scripts—the mon script will appear on the same line as the Lustre resource.**
5. **Restart the Heartbeat software.**

The trap should appear in syslog:

```
Apr 26 13:45:38 d2_q_0 mon[3000]: trap trap 1 from 192.168.0.150 \
for alpha-ost lustre_a, status 255
```

8.6 Using MMP

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

The backing filesystem for Lustre, `ldiskfs`, supports the MMP mechanism. A block in the filesystem is updated by a `kmmpd` daemon at one second intervals, and a monotonically increasing sequence number is written in this block. If the filesystem is cleanly unmounted, then a special "clean" sequence is written in this block. When mounting a filesystem, `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 filesystem, there may be a 'race'

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

Note – The MMP feature is automatically enabled by `mkfs.lustre` for new filesystems at format time if failover is being used and the kernel and e2fsprogs support it. Otherwise, it is up to the administrator to manually enable this feature when the filesystem 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`
 - To manually disable MMP: `tune2fs -O ^mmp <device>`
 - To manually enable MMP: `tune2fs -) mmp <device>`
-

If `ldiskfs` detects that a filesystem 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](#)).

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
- llnutls -> 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 symlinks from /etc/init.d/lustre to /etc/init.d/<resource_name>

These links must exist before running the conversion script.

Placing these scripts in /etc/init.d/ causes the conversion script to identify the script as type lsb, providing more flexibility for script parameters. Scripts in /etc/ha.d/resource.d are considered type heartbeat, with more restrictions.

2. 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/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
```

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
```

3. 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 fallback 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 filesystem:

```
$ mkfs.lustre --fsname=spfs --ost --failnode=oss162 \
--mgtnode=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/heartbeat/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 block momentarily.

8.8 Considerations with Failover Software and Solutions

The failover mechanisms used by Lustre and tools such as Heartbeat 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 cannot be a 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. This is true for every HA monitor, not just the Lustre `health_check`.

Indeed, 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 ($\text{obd_timeout} / 20, 5$).

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 maximum amount of disk space a user or group can consume 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 have been set, the quota of the creator's group is checked. If a quota exists for that group, then the file size is counted towards the group's quota. If no quota exists for that group, then the owner's user quota is checked before the file is written. In a similar manner, 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:

- Quota is administered via the `lfs` command
- Quota is distributed (as Lustre is a distributed filesystem), which has several ramifications
- Quota is 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 filesystem and turned on automatically at mount.

9.1.1 Configuring Disk Quotas

Enabling Quotas

1. If you have re-compiled 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 filesystem on the client and verify that the `lquota` module has loaded properly by using the `lsmod` command.

```
$ lsmod
[root@oss161 ~]# lsmod
Module                Size          Used by
obdfilter              220532         1
fsfilt_ldiskfs         52228          1
ost                    96712          1
mgc                    60384          1
ldiskfs                186896         2 fsfilt_ldiskfs
lustre                 401744          0
lov                    289064         1 lustre
lquota                 107048         4 obdfilter
mdc                    95016          1 lustre
ksocklnd               111812         1
```

The Lustre `mount` command no longer recognizes the `usrquota` and `grpquota` options. If they were previously specified, remove them from `/etc/fstab`.

When quota is enabled on the filesystem, it is automatically enabled for all filesystem clients

Note – Lustre with Linux kernel 2.4 does not support quotas.

9.1.2 Creating Quota Files and Quota Administration

Once each quota-enabled filesystem is remounted, it is capable of working with disk quotas. However, the filesystem 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 – Lustre MUST not be performing any write operations when `lfs quotacheck` is run. 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. The following options are available:

- `u` — to check the user disk quota information
- `g` — to check the group disk quota information

It also checks all the objects on all OSTs and the MDS to sum up for every UID/GID. It reads all Lustre metadata and recomputes the number of blocks/inodes that each UID/GID has used. If there are many files in Lustre, it may take a long time to complete. If quota breaks for any reason, it brings everything back.

The `lfs` command now includes these other command options for working with quotas:

- `quotaon` — announces to the system that disk quotas should be enabled on one or more filesystems. The filesystem quota files must be present in the root directory of the specified filesystem.
- `quotaoff` — announces to the system that the specified filesystems should have all the 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 set properly or users may experience unnecessary failures. The filesystem block quota is divided up among the OSTs within the filesystem. 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 filesystem block quota is divided up among the OSTs and the MDS within the file system. Only the MDS uses the filesystem 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 filesystem 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	9920	102400			1041	0	5000	
lustre-OST0000_UUID		103204*				0	102400	
lustre-OST0001_UUID		103236*				0	102400	

The total quota of 30,920 is allotted to user bob, which is further distributed to two OSTs and one MDS with a 102,400 block quota.

Note – Values appended with “*” show the limit that has been over-used (exceeding the quota), and receives the 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 within which users are allowed to exceed their soft limit. There are four types of grace period:

- 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, you can use `lfs df -i` (and also `/proc/fs/lustre/*/*/filestotal`). This command may report a lower 'total inode count' than the number of inodes actually in the filesystem. If the underlying filesystem has fewer free blocks than inodes, then the total inode count for that filesystem 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.

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 filesystem's total inode count. Instead, you should know (accurately), the free inode count and the used inode count for a filesystem. 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
```


RAID

This chapter describes RAID storage and Lustre, and includes the following sections:

- [Considerations for Backend Storage](#)
- [Insights into Disk Performance Measurement](#)
- [Creating an External Journal](#)

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 Reliability

A quick calculation (shown below), makes it clear that without further redundancy, RAID5 is not acceptable for large clusters and RAID6 is a must.

Calculation

Take a 1 PB filesystem (2000 disks of 500GB capacity). The MTF¹ of a disk is about 1000 days and repair time at 10% of disk bandwidth is close to 1 day (500 GB at 5 MB/sec = 100,000 sec = 1 day). This means that the expected failure rate is $2000/1000 = 2$ disks per day.

If we have a RAID5 stripe that is ~10 wide, then during 1 day of rebuilding, the chance that a second disk in the same array fails is about $9 / 1000 \approx 1/100$. This means that in the expected period of 50 days, a double failure in a RAID5 stripe leads to data loss.

So RAID6 or another double parity algorithm is necessary for OST storage. For the MDS, we recommend RAID0+1 storage.

10.1.2 Selecting Storage for the MDS and OSS

The MDS does a large amount of small writes. For this reason, we recommend that you use RAID1 storage. Building RAID1 Linux MD devices and striping over these devices with LVM makes it easy to create an MDS filesystem of 1-2 TB (for example, with 4 or 8 500 GB disks).

It is considered mandatory that you use disk monitoring software, so rebuilds happen without any delay. We recommend backups of the metadata filesystems. This can be done with LVM snapshots or using raw partition backups.

We also recommend using a kernel version of 2.6.15 or later with bitmap RAID rebuild features. This reduces RAID recovery time from a rebuild to quick resynchronization.

1. Mean Time to Failure

10.1.3 Understanding Double Failures with Software and Hardware 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. Such interruptions are normally caused by an abrupt shutdown of the system.

If the array functions without disk failures, but experiences sudden power-down incidents, such interrupted writes on journal filesystems, 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 filesystems with software RAID is similar to recovery without software RAID. Using Lustre servers with disk filesystems 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 a power down is followed by a disk failure, the disk filesystem needs a filesystem check.
- 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.

In the future, a modification will be released to the disk filesystem that eliminates this requirement for a check with a feature called "journal checksums". 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 Considerations

We are currently improving the Linux software RAID code to preserve large I/O which the disk subsystems can do very efficiently. With existing RAID code software, RAID performs equally with all stride sizes, but we expect that fairly large stride sizes will prove advantageous when these fixes are implemented.

10.1.5 Formatting

When formatting a filesystem on a RAID device, it is beneficial to specify additional parameters at the time of formatting. This ensures that the filesystem is optimized for the underlying disk geometry. Use the `--mkfsoptions` parameter to specify these options in the Lustre configuration.

For RAID5, RAID6, RAID1+0 storage, specifying the `-E stride={stride_size}` option improves the layout of the filesystem metadata ensuring that no single disk contains all of the allocation bitmaps. The `stride_size` 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 filesystems.

Note – It is better to have the MDS on RAID1+0 than on RAID5 or RAID6.

For more information on how to override the defaults while formatting MDS or OST filesystems, see [Options to Format MDT and OST Filesystems](#).

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.

Benchmark all disks individually. We have frequently encountered situations where drive performance was not consistent for all devices in the array.

Note – On a Thumper system with the out-of-tree Marvell driver, disk ordering is not preserved between boots (but controller ordering is). Perform a `sgp_dd` survey and create arrays without rebooting in-between.

- 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.

- By default, the maximum size of a request is quite small.

To properly handle I/O request sizes greater than 256 KB, the current Linux kernel needs either a driver patch or several changes in the block layer defaults, namely MAX_SECTORS, MAX_PHYS_SEGMENTS and MAX_HW_SEGMENTS.

- 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 further information on I/O schedulers, refer to:

<http://www.linuxjournal.com/article/6931>

<http://www.redhat.com/magazine/008jun05/features/schedulers/>

- Use the proper block device with `sgp_dd` (sgX versus sdX)

```
size 1048576K rsz 128 crg 8 thr 32 read 20.02 MB/s
```

```
size 1048576K rsz 128 crg 8 thr 32 read 56.72 MB/s
```

The above outputs were achieved on the same disk with the same parameters for `sgp_dd`. The only difference between them is that, in the first case, `/dev/sda` was used; in the second case, `/dev/sg0` was used. `sgX` is a special interface that bypasses the block layer and the I/O scheduler, but sends the SCSI commands directly to a drive. `sdX` is a regular block device, and the requests go through the block layer and the I/O scheduler. Using different I/O schedulers does not change the numbers.

Tip – Try to avoid sync writes; probably subsequent write would make the stripe full and no reads will be needed. Try to configure RAID5 and the application so that most of the writes are full-stripe and stripe-aligned.

- `NR_STRIPE` in RAID5 (Linux kernel parameter)

This is the size of the internal cache that RAID5 uses for all operations. If many processes are doing I/O, we suggest that you increase this number. In newer kernels, there is a module parameter to tune it.

- Do not put an ext3 journal onto RAID5.

As the journal is written linearly and synchronously, in most cases writes do not fill whole stripes and RAID5 has to read parities.

- Suggested MD device setups for maximum performance:

MDT

RAID1 with an internal journal and two disks from different controllers.

If you need larger MDTs, create two equally-sized RAID0 arrays from multiple disks. Then create a RAID1 array from these two arrays. Using RAID10 directly requires a newer mdadm² than the one shipped with RHEL 4. You should also be able to use LVM instead of RAID0, although this has not been tested.

OST

Filesystem: RAID5 with 6 disks, each from a different controller.

External journal: RAID1 with two partitions of 400 MB (or more), each from disks on different controllers.

```
$ mkfs.lustre ... --mkfsoptions "-j -J device=/dev/mdX"
```

To set up the journal device (/dev/mdX), run:

```
$ 'mke2fs -O journal_dev -b 4096 /dev/mdX'
```

Then run `--reformat`:

```
$ mkfs.lustre ... --reformat... --mkfsoptions "-j -J device= /  
/dev/mdX"
```

You can create a root filesystem, swap, etc. the RAID1 array of partitions on any two remaining disks. The remaining space on the OST journal disk can be used for this.

Note – We have not tested RAID1 of swap.

■ rsz in `sgp_dd`

It must equal **<chunksize> * (disks-1)**.

You should pass `stripe=N`, `extents`, `mballoc` as the `mountfs` option for the OSS where:

N=<chunksize> * (disks-1)/pagesize.

When setting up the array, select **<chunksize>** such that **<chunksize> * (number of active disks)** is close to 1 MB, but not larger.

2. The tool that administers software RAID on Linux.

- Run `fsck` on a combined power and disk failure (RAID arrays).
In the event of a combined power and disk failure, you must run `fsck` on an array (owing to potential write-consistency issues).

Note – You should be able to automate this in `rc.sysinit` by detecting degraded arrays.

- Consider using writeback cache
The table below shows the benefits of writeback cache.

<tablewidth="" tablealign="">OSTs	external journal	writeback cache	write performance
1	no	no	45.93 MB/sec
1	yes (1 drive)	no	117.55 MB/sec
1	yes (2 drives)	no	130.52 MB/sec
6	no	no	246.15 MB/sec
6	no	yes	330.98 MB/sec
6	yes (1 drive)	no	239.83 MB/sec
6	yes (2 drives)	no	414.94 MB/sec



Caution – Writeback cache is dangerous from a data integrity point of view. You should carefully consider whether the benefits of using writeback cache outweigh the risks.

10.2.1 Sample Graphs

10.2.1.1 Graphs for Write Performance

FIGURE 10-1 Write - RAID0, 64K chunks, 6 spindles

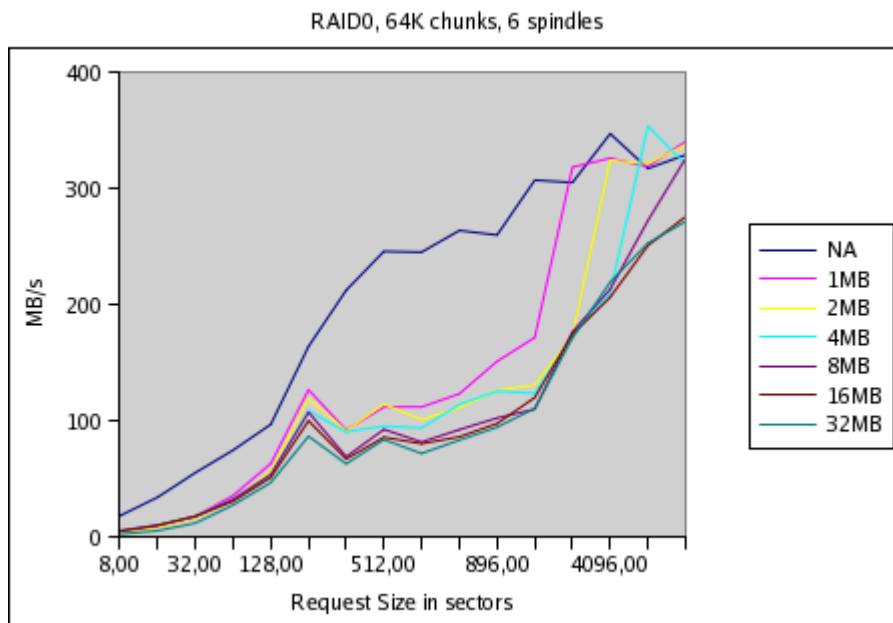
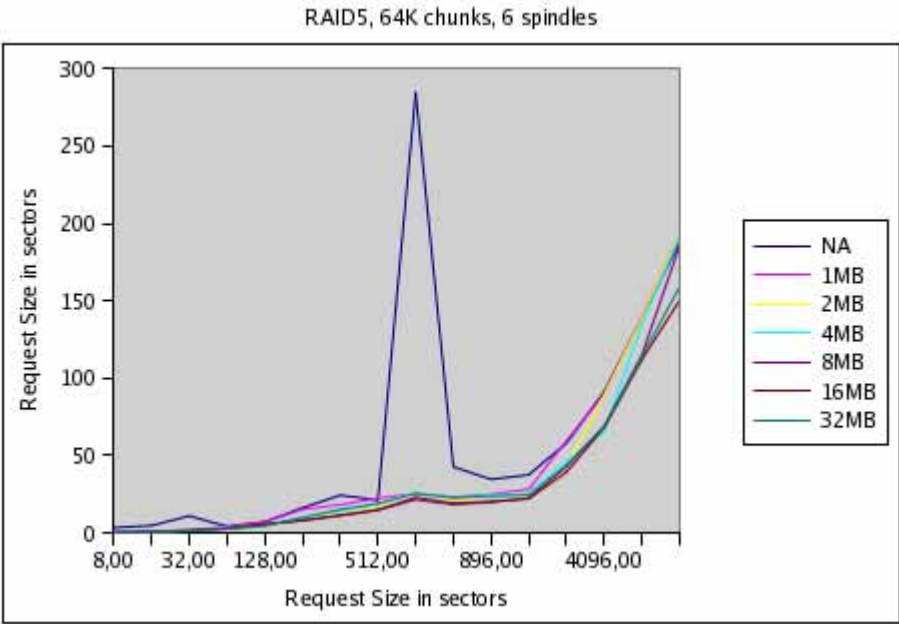


FIGURE 10-2 Write - RAID0, 64K chunks, 6 spindles



10.2.1.2 Graphs for Read Performance

FIGURE 10-3 Read - RAID0, 64K chunks, 6 spindles

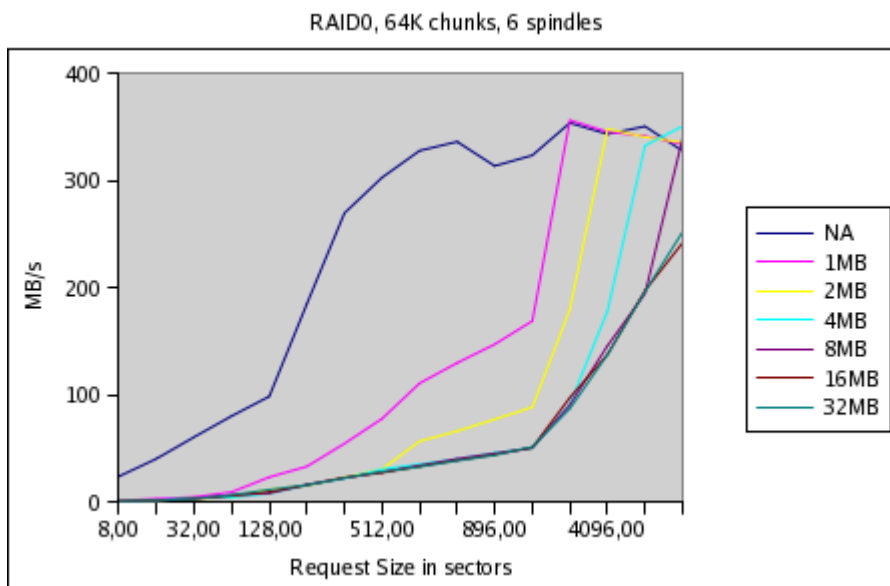
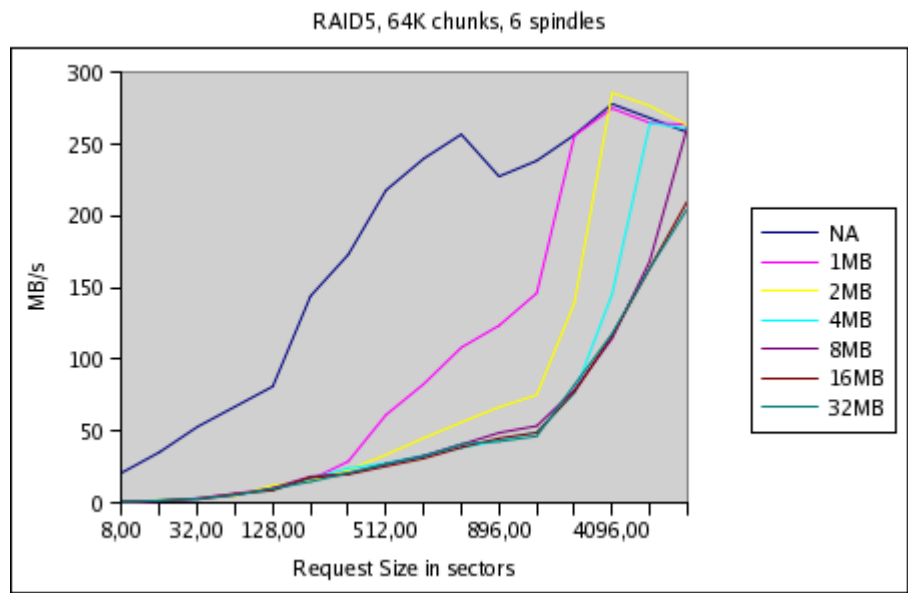


FIGURE 10-4 Read - RAID5, 64K chunks, 6 spindles



10.3 Creating an External Journal

To create an external journal:

1. **Format the Lustre MDT/MGS to use an external journal device. Format the sdb with:**

```
$ mkfs.lustre --mdt --mgs --mkfsoptions="-J device=/dev/sdb"
/dev/sda
```

2. **Format a Lustre target sda to use an external journal on sdb.**

```
$ mkfs.lustre --mdt --mgs --mkfsoptions="-J device=/dev/sdb" \
/dev/sda
Permanent disk data:
Target:          lustre-MDTffff
Index:           unassigned
Lustre FS:       lustre
Mount type:      ldiskfs
Flags:           0x75
(MDT MGS needs_index first_time update )
Persistent mount opts: errors=remount- ro,iopen_nopriv,user_xattr
Parameters:

device size = 4096MB
formatting backing filesystem ldiskfs on /dev/sda
target name      lustre-MDTffff
4k blocks        0
options          -J device=/dev/sdb -i 4096 -I 512 -q -O dir_index -F
mkfs_cmd = mkfs.ext2 -j -b 4096 -L lustre-MDTffff -J device= \
/dev/sdb -i 4096 -I
512 -q -O dir_index -F /dev/sda
Writing CONFIGS/mountdata
```

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

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

4. **Format the Lustre OSS to use an external journal device. Format the sdb with:**

```
$ mke2fs -b 4096 -O journal_dev /dev/sdb
```


5. Format a Lustre target sda to use an external journal on sdb.

```
# mkfs.lustre --ost --reformat --mkfsoptions="-J device=/dev/sdb"\
--mgsgnode=mds16@tcp0 /dev/sda
Permanent disk data:
Target:      lustre-OSTffff
Index:       unassigned
Lustre FS:   lustre
Mount type:  ldiskfs
Flags:       0x72
(OST needs_index first_time update )
Persistent mount opts: errors=remount-ro, extents, mballoc
Parameters: mgsgnode=192.168.16.21@tcp

device size = 4096MB
formatting backing filesystem ldiskfs on /dev/sda
target name lustre-OSTffff
4k blocks      0
options        -J device=/dev/sdb -i 16384 -I 256 -q -O dir_index -F
mkfs_cmd = mkfs.ext2 -j -b 4096 -L lustre-OSTffff -J device= \
/dev/sdb -i \ 16384 -I
256 -q -O dir_index -F /dev/sda
Writing CONFIGS/mountdata
```

6. Mount Lustre on the client, run:

```
$ mkdir -p /mnt/testfs
$ mount -t lustre cfs21@tcp0:/testfs /mnt/testfs
```


Kerberos

This chapter describes how to use Kerberos with Lustre and includes the following sections:

- [What is Kerberos?](#)
- [Lustre Setup with Kerberos](#)

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 Lustre 1.8 feature that is not available in version 1.6 Do NOT attempt to use Kerberos with Lustre until 1.8 is released.

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.

For the latest update, go to: <http://www.clusterfs.com>

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/libgssapi-0.10.tar.gz
```
 - keyutils

11.2.1.3 Configuring Lustre for Kerberos

To configure Lustre for Kerberos:

1. **Configure the client nodes. For each client node, create a lustre_root principal, and generate and install the keytab.**

```
kadmin> addprinc -randkey lustre_root@REALM
kadmin> ktadd -e des-cbc-md5:normal lustre_root@REALM
```

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.

2. Configure the MDT 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 des-cbc-md5:normal
lustre_mds/mdthost.domain@REALM
```

3. Configure the OST nodes. For each OST node, create a `lustre_oss` principal, and generate and install the keytab.

```
kadmin> addprinc -randkey lustre_oss/osthost.domain@REALM
kadmin> ktadd -e des-cbc-md5:normal lustre_oss/osthost.domain@REALM
```

Lustre supports almost all useful encryption types which are supported by MIT Kerberos 5:

- `des-cbc-crc`
- `des-cbc-md5`
- `des3-hmac-sha1`
- `aes128-cts`
- `aes256-cts`
- `arcfour-hmac-md5`

Note – Encryption Type or enctype is an identifier that specifies the encryption algorithm, mode and hash algorithms. Keys in Kerberos have an associated enctype to identify the cryptographic algorithm and mode to be used when performing cryptographic operations with the key. It is important that the encetypes requested by the client are actually supported on the system hosting the client. This is the case if the defaults that control encetypes are not overridden.

Note – For MIT Kerberos 1.3.x, only `des-cbc-md5` works because of a known issue between `libssapi` and the Kerberos library. The `host.domain` should be the FQDN in your network; otherwise the server might not recognize the GSS request.

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@CLUSTERFS.COM
kadmin> ktadd -e des-cbc-md5:normal lustre_root@ CLUSTERFS.COM
```

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 MDT nodes. For each MDT node, create a `lustre_mds` principal, and generate and install the keytab.**

```
kadmin> addprinc -randkey lustre_mds/mdthost.domain@CLUSTERFS.COM
kadmin> ktadd -e des-cbc-md5:normal \
lustre_mds/mdthost.domain@CLUSTERFS.COM
```

6. **Configure the OST nodes. For each OST node, create a `lustre_oss` principal, and generate and install the keytab.**

```
kadmin> addprinc -randkey lustre_oss/osthost.domain@CLUSTERFS.COM
kadmin> ktadd -e des-cbc-md5:normal lustre_oss/osthost.domain@
CLUSTERFS.COM
```

For more detailed information on installing 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, add 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 we turn on GSS between MDT-OST or MDT-MDT, GSS treats MDT as a client. Hence, `lgssd` should be running on MDT.

There are two types of GSS daemons: `lgssd` and `lsvcgssd`. Before starting Lustre, make sure they are running on each node before starting Lustre:

- OST: `lsvcgssd`
- MDT: `lsvcgssd`
- CLI: none

Note – 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 flavor
- Security flavor
- Customized flavor

Select a flavor depending on your priorities and preferences.

11.2.2.1 Basic Flavor

Currently, we support four basic flavors: *null*, *plain*, *krb5i*, and *krb5p*, described in [TABLE 11-1](#).

TABLE 11-1 Basic Supported Kerberos Flavors

Basic Flavor	Authentication	RPC Message Protection	Bulk Data Protection	Remarks
<i>null</i>	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).
<i>plain</i>	N/A	null	null	Carries checksum (So only protects data mutating during transfer, does NOT guarantee the genuine author as there is no actual authentication).
<i>krb5i</i>	GSS/Kerberos5	integrity	integrity [SHA1]	RPC message integrity protection algorithm is determined by actual Kerberos algorithms in use; heavy performance overhead.
<i>krb5p</i>	GSS/Kerberos5	privacy	privacy [SHA1/ARC4]	RPC message privacy protection algorithm is determined by actual Kerberos algorithms in use; considerable performance overhead.

11.2.2.2 Security Flavor

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: RPC message and BULK data. You can set any one of the parts in one of the following three modes:

- ***null*** – No protection
- ***integrity*** – Data integrity protection (checksum or signature)
- ***privacy*** – Data privacy protection (encryption)

11.2.2.3 Customized Flavor

Generally, you do not need the customized flavor; basic flavor is sufficient for regular usage. But, you can customize the flavor string to some extent. The usual format of a flavor string is:

```
Qbasic_flavor[-bulk{nip}[:checksum_alg[/encryption_alg]]]
```

Here are some examples of how to use customized flavors:

- ***plain-bulki***: Use plain on RPC message (offering null protection), but add checksum protection on the bulk transfer.
- ***krb5i-bulkn***: Use krb5i on RPC message, but do not protect the bulk transfer.
- ***krb5p-bulki***: Use krb5p on RPC message, but protect data integrity of the bulk transfer.
- ***krb5p-bulkp:sha512/arc4***: Use krb5p on RPC message, and protect data privacy of the bulk transfer by algorithm SHA512 and ARC4.

Currently, Lustre supports following bulk data crypto algorithms:

- Checksum:
 - ocrc32
 - omd5
 - osha1/sha256/sha384/sha512
- Encryption:
 - oarc4

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 Authenticating Normal Users

On client nodes, a non-root user needs kinit before accessing Lustre, just like other Kerberized applications. You can destroy the established security contexts before logging out by `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 Lustre kernel.

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](#)

12.1 Network Bonding

Bonding, also known as link aggregation, trunking and port trunking, is a method of aggregating multiple physical network links into a single logical link for increased bandwidth.

Several different types of bonding are supported in Linux. All these types are referred to as “modes,” and use the bonding kernel module.

Modes 0 to 3 provide support for load balancing and fault tolerance by using multiple interfaces. Mode 4 aggregates a group of interfaces into a single virtual interface where all members of the group share the same speed and duplex settings. This mode is described under IEEE spec 802.3ad, and it is referred to as either “mode 4” or “802.3ad.”

(802.3ad refers to mode 4 only. The detail is contained in Clause 43 of the IEEE 8 - the larger 802.3 specification. For more information, consult IEEE.)

12.2 Requirements

The most basic requirement for successful bonding is that both endpoints of the connection must support bonding. In a normal case, the non-server endpoint is a switch. (Two systems connected via crossover cables can also use bonding.) Any switch used must explicitly support 802.3ad Dynamic Link Aggregation.

The kernel must also support bonding. All supported Lustre kernels have bonding functionality. The network driver for the interfaces to be bonded must have the ethtool support. To determine slave speed and duplex settings, ethtool support is necessary. All recent network drivers implement it.

To verify that your interface supports ethtool, run:

```
# which ethtool
/sbin/ethtool

# ethtool eth0
Settings for eth0:
    Supported ports: [ TP MII ]
    Supported link modes:   10baseT/Half 10baseT/Full/
                           100baseT/Half 100baseT/Full
    Supports auto-negotiation: Yes
    Advertised link modes:  10baseT/Half 10baseT/Full
                           100baseT/Half 100baseT/Full
    Advertised auto-negotiation: Yes
    Speed: 100Mb/s
    Duplex: Full
    Port: MII
    PHYAD: 1
    Transceiver: internal
    Auto-negotiation: on
    Supports Wake-on: pumbg
    Wake-on: d
    Current message level: 0x00000001 (1)
    Link detected: yes
```



```
# ethtool eth1

Settings for eth1:
    Supported ports: [ TP MII ]
    Supported link modes:   10baseT/Half 10baseT/Full
                           100baseT/Half 100baseT/Full
    Supports auto-negotiation: Yes
    Advertised link modes:  10baseT/Half 10baseT/Full
                           100baseT/Half 100baseT/Full
    Advertised auto-negotiation: Yes
    Speed: 100Mb/s
    Duplex: Full
    Port: MII
    PHYAD: 32
    Transceiver: internal
    Auto-negotiation: on
    Supports Wake-on: pumbg
    Wake-on: d
    Current message level: 0x00000007 (7)
    Link detected: yes

To quickly check whether your kernel supports bonding, run:
# grep ifenslave /sbin/ifup
# which ifenslave
/sbin/ifenslave
```

Note – Bonding and ethtool have been available since 2000. All Lustre-supported kernels include this functionality.

12.3 Using Lustre with Multiple NICs versus Bonding NICs

Lustre can use multiple NICs without bonding. There is a difference in performance when Lustre uses multiple NICs versus when it uses bonding NICs.

Whether an aggregated link actually yields a performance improvement proportional to the number of links provided, depends on network traffic patterns and the algorithm used by the devices to distribute frames among aggregated links. Performance with bonding depends on:

- Out-of-order packet delivery

This can trigger TCP congestion control. To avoid this, some bonding drivers restrict a single TCP conversation to a single adapter within the bonded group.

- Load balancing between devices in the bonded group.

Consider a scenario with a two CPU node with two NICs. If the NICs are bonded, Lustre establishes a single bundle of sockets to each peer. Since `ksocklnd` bind sockets to CPUs, only one CPU moves data in and out of the socket for a uni-directional data flow to each peer. If the NICs are not bonded, Lustre establishes two bundles of sockets to the peer. Since `ksocklnd` spreads traffic between sockets, and sockets between CPUs, both CPUs move data.

12.4 Bonding Module Parameters

Bonding module parameters control various aspects of bonding.

Outgoing traffic is mapped across the slave interfaces according to the transmit hash policy. For Lustre, we recommend that you set the `xmit_hash_policy` option to the `layer3+4` option for bonding. This policy uses upper layer protocol information if available to generate the hash. This allows traffic to a particular network peer to span multiple slaves, although a single connection does not span multiple slaves.

```
$ xmit_hash_policy=layer3+4
```

The `miimon` option enables users to monitor the link status. (The parameter is a time interval in milliseconds.) It makes an interface failure transparent to avoid serious network degradation during link failures. A reasonable default setting is 100 milliseconds; run:

```
$ miimon=100
```

For a busy network, increase the timeout.

12.5 Setting Up Bonding

To set up bonding:

1. **Create a virtual 'bond' interface by creating a configuration file in:**

```
/etc/sysconfig/network-scripts/ # vi /etc/sysconfig/ \  
network-scripts/ifcfg-bond0
```

2. **Append the following lines to the file.**

```
DEVICE=bond0  
IPADDR=192.168.10.79 # Use the free IP Address of your network  
NETWORK=192.168.10.0  
NETMASK=255.255.255.0  
USERCTL=no  
BOOTPROTO=none  
ONBOOT=yes
```

3. Attach one or more slave interfaces to the bond interface. Modify the eth0 and eth1 configuration files (using a VI text editor).

a. Use the VI text editor to open the eth0 configuration file.

```
# vi /etc/sysconfig/network-scripts/ifcfg-eth0
```

b. Modify/append the eth0 file as follows:

```
DEVICE=eth0
USERCTL=no
ONBOOT=yes
MASTER=bond0
SLAVE=yes
BOOTPROTO=none
```

c. Use the VI text editor to open the eth1 configuration file.

```
# vi /etc/sysconfig/network-scripts/ifcfg-eth1
```

d. Modify/append the eth1 file as follows:

```
DEVICE=eth1
USERCTL=no
ONBOOT=yes
MASTER=bond0
SLAVE=yes
BOOTPROTO=none
```

4. Set up the bond interface and its options in /etc/modprobe.conf. Start the slave interfaces by your normal network method.

```
# vi /etc/modprobe.conf
```

a. Append the following lines to the file.

```
alias bond0 bonding
options bond0 mode=balance-alb miimon=100
```

b. Load the bonding module.

```
# modprobe bonding
# ifconfig bond0 up
# ifenslave bond0 eth0 eth1
```

5. Start/restart the slave interfaces (using your normal network method).

Note – You must modprobe the bonding module for each bonded interface. If you wish to create bond0 and bond1, two entries in `modprobe.conf` are required.

The examples below are from RedHat systems. For setup use:
`/etc/sysconfig/networking-scripts/ifcfg-*` The OSDL website referenced below includes detailed instructions for other configuration methods, instructions to use DHCP with bonding, and other setup details. We strongly recommend you use this website.

<http://linux-net.osdl.org/index.php/Bonding>

6. Check `/proc/net/bonding` to determine status on bonding. There should be a file there for each bond interface.

```
# cat /proc/net/bonding/bond0
Ethernet Channel Bonding Driver: v3.0.3 (March 23, 2006)

Bonding Mode: load balancing (round-robin)
MII Status: up
MII Polling Interval (ms): 0
Up Delay (ms): 0
Down Delay (ms): 0

Slave Interface: eth0
MII Status: up
Link Failure Count: 0
Permanent HW addr: 4c:00:10:ac:61:e0

Slave Interface: eth1
MII Status: up
Link Failure Count: 0
Permanent HW addr: 00:14:2a:7c:40:1d
```

7. Use ethtool or ifconfig to check the interface state. ifconfig lists the first bonded interface as “bond0.”

```
ifconfig
bond0      Link encap:Ethernet  HWaddr 4C:00:10:AC:61:E0
            inet addr:192.168.10.79  Bcast:192.168.10.255  \
            Mask:255.255.255.0
            inet6 addr: fe80::4e00:10ff:feac:61e0/64 Scope:Link
            UP BROADCAST RUNNING MASTER MULTICAST  MTU:1500  \
            Metric:1
            RX packets:3091 errors:0 dropped:0 overruns:0 frame:0
            TX packets:880 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:0
            RX bytes:314203 (306.8 KiB)  TX bytes:129834 (126.7 KiB)

eth0       Link encap:Ethernet  HWaddr 4C:00:10:AC:61:E0
            inet6 addr: fe80::4e00:10ff:feac:61e0/64 Scope:Link
            UP BROADCAST RUNNING SLAVE MULTICAST  MTU:1500  Metric:1
            RX packets:1581 errors:0 dropped:0 overruns:0 frame:0
            TX packets:448 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:1000
            RX bytes:162084 (158.2 KiB)  TX bytes:67245 (65.6 KiB)
            Interrupt:193 Base address:0x8c00

eth1       Link encap:Ethernet  HWaddr 4C:00:10:AC:61:E0
            inet6 addr: fe80::4e00:10ff:feac:61e0/64 Scope:Link
            UP BROADCAST RUNNING SLAVE MULTICAST  MTU:1500  Metric:1
            RX packets:1513 errors:0 dropped:0 overruns:0 frame:0
            TX packets:444 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:1000
            RX bytes:152299 (148.7 KiB)  TX bytes:64517 (63.0 KiB)
            Interrupt:185 Base address:0x6000
```

12.5.1 Examples

This is an example of `modprobe.conf` for bonding Ethernet interfaces `eth1` and `eth2` to `bond0`:

```
# cat /etc/modprobe.conf
alias eth0 8139too
alias scsi_hostadapter sata_via
alias scsi_hostadapter1 usb-storage
alias snd-card-0 snd-via82xx
options snd-card-0 index=0
options snd-via82xx index=0
alias bond0 bonding
options bond0 mode=balance-alb miimon=100
options lnet networks=tcp
alias eth1 via-rhine

# cat /etc/sysconfig/network-scripts/ifcfg-bond0
DEVICE=bond0
BOOTPROTO=none
NETMASK=255.255.255.0
IPADDR=192.168.10.79 # (Assign here the IP of the bonded interface.)
ONBOOT=yes
USERCTL=no

ifcfg-ethx
# cat /etc/sysconfig/network-scripts/ifcfg-eth0
TYPE=Ethernet
DEVICE=eth0
HWADDR=4c:00:10:ac:61:e0
BOOTPROTO=none
ONBOOT=yes
USERCTL=no
IPV6INIT=no
PEERDNS=yes
MASTER=bond0
SLAVE=yes
```

In the following example, the bond0 interface is the master (MASTER) while eth0 and eth1 are slaves (SLAVE).

Note – All slaves of bond0 have the same MAC address (Hwaddr) – bond0. All modes, except TLB and ALB, have this MAC address. TLB and ALB require a unique MAC address for each slave.

```
$ /sbin/ifconfig
```

```
bond0Link encap:EthernetHwaddr 00:C0:F0:1F:37:B4
inet addr:XXX.XXX.XXX.YYY Bcast:XXX.XXX.XXX.255 Mask:255.255.252.0
UP BROADCAST RUNNING MASTER MULTICAST MTU:1500 Metric:1
RX packets:7224794 errors:0 dropped:0 overruns:0 frame:0
TX packets:3286647 errors:1 dropped:0 overruns:1 carrier:0
collisions:0 txqueuelen:0
```

```
eth0Link encap:EthernetHwaddr 00:C0:F0:1F:37:B4
inet addr:XXX.XXX.XXX.YYY Bcast:XXX.XXX.XXX.255 Mask:255.255.252.0
UP BROADCAST RUNNING SLAVE MULTICAST MTU:1500 Metric:1
RX packets:3573025 errors:0 dropped:0 overruns:0 frame:0
TX packets:1643167 errors:1 dropped:0 overruns:1 carrier:0
collisions:0 txqueuelen:100
Interrupt:10 Base address:0x1080
```

```
eth1Link encap:EthernetHwaddr 00:C0:F0:1F:37:B4
inet addr:XXX.XXX.XXX.YYY Bcast:XXX.XXX.XXX.255 Mask:255.255.252.0
UP BROADCAST RUNNING SLAVE MULTICAST MTU:1500 Metric:1
RX packets:3651769 errors:0 dropped:0 overruns:0 frame:0
TX packets:1643480 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:100
Interrupt:9 Base address:0x1400
```

12.6 Configuring Lustre with Bonding

Lustre uses the IP address of the bonded interfaces and requires no special configuration. It treats the bonded interface as a regular TCP/IP interface. If needed, specify “bond0” using the Lustre networks parameter in `/etc/modprobe`

```
options lnet networks=tcp(bond0)
```

12.6.1 Bonding References

We recommend the following bonding references:

In the Linux kernel source tree, see `documentation/networking/bonding.txt`

<http://linux-ip.net/html/ether-bonding.html>

<http://www.sourceforge.net/projects/bonding>

This is the bonding SourceForge website:

<http://linux-net.osdl.org/index.php/Bonding>

This is the most extensive reference and we highly recommend it. This website includes explanations of more complicated setups, including the use of DHCP with bonding.

Upgrading Lustre

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

- [Lustre Interoperability](#)
- [Upgrading from Version 1.4.11 to Version 1.6.3](#)
- [Downgrading Lustre from Version 1.6.3 to Version 1.4.11](#)

13.1 Lustre Interoperability

For Lustre 1.6.3, the following upgrades are supported:

- Lustre 1.4.11 (latest 1.4.x version) to Lustre 1.6.4 (latest 1.6.x version).
- One minor version to the next (for example, 1.6.3 > 1.6.4 and 1.4.10 > 1.4.11).

For Lustre 1.6.3, downgrades in the same ranges are supported.

- If you upgrade from Lustre 1.4.11 > 1.6.4, you can downgrade to version 1.4.11.
- If you upgrade from Lustre 1.6.2 > 1.6.4, you can downgrade to version 1.6.3.



Caution – A fresh installation of Lustre 1.6.4 is not guaranteed to be downgradable to an earlier Lustre version.

13.2 Upgrading from Version 1.4.11 to Version 1.6.3

Use the procedures in this chapter to upgrade Lustre version 1.4.11 to version 1.6.3.

13.2.1 Upgrade Requirements

Remember the following important points before upgrading Lustre.

Upgrade MDT before OSTs. The upgrade procedure is:

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

Upgrade can be done across a failover pair, in which case the upgrade procedure is:

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

The filesystem name must be less than or equal to 8 characters (so it fits on the disk label).

13.2.2 Supported Upgrade Paths

The following Lustre upgrade paths are supported.

Entire filesystem 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 filesystem never goes down. This type of upgrade limits your ability to change certain parameters.
- The entire filesystem 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 filesystem which was created by a new MDT.

13.2.3 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 <mgssid>:/<fsname> <mountpoint>
```

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

13.2.4 Upgrading a Single Filesystem

`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 filesystem, 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 filesystem 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# tuneufs.lustre --ost --fsname=lustre --mgsnode=mds
/dev/sda4
checking for existing Lustre data: found last_rcvd
tuneufs.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 upgradel.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 upgradel.4 )
Persistent mount opts: errors=remount-ro,extents,malloc
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
```


13.2.5 Upgrading Multiple Filesystems with a Shared MGS

The upgrade order is: MGS first, then for any single filesystem the MDT must be upgraded and mounted, and then the OSTs for that filesystem. If the MGS is co-located with the MDT, the old config 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 filesystems), 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# tuneufs.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 filesystem.

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

10. Upgrade other MDTs in a similar manner. Keep in mind:

- 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).

13.3 Downgrading Lustre from Version 1.6.3 to Version 1.4.11

This section describes how to downgrade Lustre version 1.6.3 to version 1.4.11.

13.3.1 Downgrade Requirements

- The filesystem must have been upgraded from 1.4.x. In other words, a filesystem created or reformatted under 1.6 cannot be downgraded.
- Any new OSTs that were dynamically added to the filesystem 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 filesystems that the MGS serves.

13.3.2 Downgrading a Filesystem

To downgrade a filesystem:

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 filesystem was upgraded are lost.

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.conf.**

```
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 is 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 (`lustrePortalsCatastropheTrap`), and detection of various OBD-specific healthchecks (`lustreOBDUnhealthyTrap`).

Backup and Restore

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

- [Lustre Backups](#)
- [Restoring from a File-level Backup](#)

15.1 Lustre Backups

Lustre provides filesystem backups at several levels.

15.1.1 Client Filesystem-level Backups

It is possible to get a backup of Lustre filesystems from a client (or many clients working parallel in different directories) with the help of user-level backup tools tar, cpio, amanda, and many other enterprise-level backup utilities. Using normal, file backup tools remains the easiest, recommended method to back up and restore data.

However, due to the large size of most Lustre filesystems, it is not always possible to get a complete backup. We recommend that you back up subsets of a filesystem. This includes subdirectories of the entire directory, filesets for a single user, files incremented by date, and so on.

15.1.2 Performing Device-level Backups

In some situations, you may need a full, device-level backup of an individual MDS or OST storage device (before replacing hardware, performing maintenance, etc.). A full device-level backup is the easiest backup method and it ensures preservation of the original data.

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 filesystem is very robust and it may be possible to recover the filesystem 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 filesystem, 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 filesystem need to be changed, or to use less space for backup.

You can mount the ext3 filesystem 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) stored in the filesystem. As the current backup tools do not properly save this data, perform the following procedure.

15.1.3.1 Backing Up an MDS File

To back up a file on the MDS:

1. **Make a mount point for the filesystem "mkdir /mnt/mds" and mount the filesystem 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 filesystem data, run:**

```
tar czvf {backup file}.tgz
```

6. **Change directory out of the mounted filesystem, run:**

```
cd -
```

7. **Unmount the filesystem, 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 filesystem, 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 filesystems, run:

```
mk2fs -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)`)

For OST filesystems, run:

```
mk2fs -j -J size=400 -I 256 -i 16384 {dev}"
```

2. Enable ext3 filesystem directory indexing.

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

3. Mount the filesystem.

- 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 filesystem mount point, run:

```
cd /mnt/mds
```

5. Restore the filesystem backup, run:

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

6. Restore the filesystem 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 filesystem is in use during the restore process, run the `fsck` tool (part of `e2fsprogs`) to ensure that the filesystem is coherent.

It is not necessary to run this tool if the backup of all device filesystems occurs at the same time after stopping the entire Lustre filesystem. After completing the filesystem should be immediately usable without running `fsck`. 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.

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 filesystem 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 – Advisory fcntl/flock/lockf locks will be available in Lustre 1.8.

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

```
ftp://ftp.lustre.org:/pub/benchmarks/posix/tet_vsxgen_2.0.tgz
```

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



Caution – Do not configure or mount a Lustre filesystem 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 filesystems 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 filesystem tests.

b. Run the test sets.

Even though it is running them on a local filesystem, 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 filesystem 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 filesystem. 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 filesystem, 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 filesystem, 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_DEBUG_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_DEBUG_FLAGS=xxxxx

The following example makes VSX output all debug messages:

VSX_DEBUG_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.

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 filesystem design, implementation, and performance.

This chapter describes benchmark suites to test Lustre, and includes the following sections:

- [Bonnie++ Benchmark](#)
- [IOR Benchmark](#)
- [IOzone Benchmark](#)

17.1 Bonnie++ Benchmark

Bonnie++ is a benchmark suite that having aim of performing a number of simple tests of hard drive and filesystem 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 filesystem performance by sequential I/O and random seeks. Bonnie++ tests filesystem 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, +++++, +++, 2
83, 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,27460,92,41450,25,21474,10,19673,60,52871,10,88.0,0,16,296
81,99,+++++,+++,30412,90,29568,99,+++++,+++,28077,82

```

17.2 IOR Benchmark

Use the IOR_Survey script to test the performance of the lustre filesystems. It uses IOR (Interleaved or Random), a script used for testing performance of parallel filesystems 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:

<ftp://ftp.llnl.gov/pub/siop/ior/>

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 filesystem 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 filesystem 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.
```

Lustre Recovery

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

- [Recovering Lustre](#)
- [Types of Failure](#)

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 filesystem in an unclear state. In other words, unsaved data may be in the client cache. To save this data, the filesystem re-starts in recovery mode and makes the clients write the data to disk.

18.1 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 filesystem 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.

18.2 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.

18.2.1 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.

18.2.2 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 filesystem. 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.

18.2.3 OST Failure

When an OST fails or is severed from the client, Lustre marks the corresponding OSC as inactive, and the LogicalObjectVolume 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.

18.2.4 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 filesystem 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 filesystem may get stuck in recovery if any servers are down or if any of servers have thrown a Lustre bug (LBUG); check `/proc/fs/lustre/health_check`.

PART III Lustre Tuning, Monitoring and Troubleshooting

The part includes chapters on how to tune, debug and troubleshoot Lustre.

Lustre I/O Kit

This chapter describes the Lustre I/O kit and PIOS performance tool, and includes the following sections:

- [Lustre I/O Kit Description and Prerequisites](#)
- [Running I/O Kit Tests](#)
- [PIOS Test Tool](#)
- [LNET Self-Test](#)

19.1 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 filesystem. 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.

19.1.1 Downloading an I/O Kit

You can download the I/O kits from:

<https://downloads.clusterfs.com/customer/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.

19.1.2 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 filesystem software
- `sg3_utils` for the `sgp_dd` utility

19.2 Running I/O Kit Tests

As mentioned above, the I/O kit contains these test tools:

- `sgpdd_survey`
- `obdfilter_survey`
- `ost_survey`

19.2.1 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.

19.2.2 obdfilter_survey

The `obdfilter_survey` script processes sequential I/O with varying numbers of threads and objects (files) by using `lctl::test_brw` 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:

Stripe F/S

Here, the script directly exercises one or more instances of `obdfilter`. The script may be running on one or more nodes, for example, when the nodes are all attached to the same multi-ported disk subsystem.

You need to tell the script the names of all `obdfilter` instances, which should already be up and running. If some instances are on different nodes, then you also need to specify their hostnames, for example, `node1:ost1`. All the `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

Here, the script drives one or more instances of obdecho via instances of `echo_client` running on one or more nodes. You need to tell the script the names of all `echo_client` instances, which should already be up and running. If some instances are on different nodes, then you also need to specify their hostnames, for example, `node1:ECHO_node1`.

Stripe F/S over the network

Here, the script drives one or more instances of obdfilter via instances of `echo_client` running on one or more nodes. As noted above, you need to tell the script the names of all `echo_client` instances, which should already be up and running. Note that the script is not scalable to hundreds of nodes since it is only intended to measure individual servers, not the scalability of the system as a whole.

Script

The script must be customized according to the components being tested and the location where it should keep its working files. Customization variables are clearly described at the beginning of the script.

19.2.2.1 Running obdfilter_survey Against a Local Disk

To run the `obdfilter_survey` script against a local disk:

- 1. Create a Lustre configuration shell script and XML, using your normal methods. You do not need to specify an MDS or LOV, but you do need to list all OSTs that you wish to test.**
- 2. On all OSS machines, run:**

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

Remember, write tests are destructive. This test should be run prior to startup of your actual Lustre filesystem. If you do this, you will not need to reformat to restart Lustre. However, if the test is terminated before completion, you may have to remove objects from the disk.

3. **Determine the obdfilter instance names on all clients. The names appear as the 4th column of `lctl dl`. 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
```

Here the obdfilter instance names are `oss01-sdb`, `oss01-sdd`, `oss02-sdi`. Since you are driving obdfilter instances directly, set the shell array variable, `ost_names`, to the names of the obdfilter instances and leave the `client_names` variable undefined. For example:

```
ost_names_str='oss01:oss01-sdb oss01:oss01-sdd oss02:oss02-sdi'\
./obdfilter-survey
```

19.2.2.2 Running obdfilter_survey Against a Network

If you are driving obdfilter or obdecho instances over the network, then you must instantiate the `echo_clients`. Set the shell array variable `client_names` to the names of the `echo_client` instances and leave the `ost_names` variable undefined.

Optionally, you can prefix any name in `ost_names` or `client_names` with the hostname that it runs on, for example, `remote_node:ost4`. If you are running remote nodes, make sure these requirements are met:

- The `custom_remote_shell()` works on your cluster.
- All pathnames that you specify in the script are mounted on the node from which `obdfilter_survey` is started and on all remote nodes.
- The `obdfilter_survey` script is installed on clients at the same location as the master node.

To run the `obdfilter_survey` script against a network:

1. **Bring up obdecho instances on the servers and echo_client instances on the clients, and run the included `echo.sh` on a node that has Lustre installed. Shell variables:**

- **SERVERS:** Set this to a list of server hostnames or hostname of the current node is used. This may be the wrong interface, so be sure to check it.

Note – `echo.sh` could probably be smarter about this.

- NETS: set this if you are using a network type other than TCP.

For example:

```
SERVERS=oss01-eth2 sh echo.sh
```

2. On the servers, start the obdecho server and verify that it is up, run:

```
$ lctl dl
0 UP obdecho ost_oss01.local ost_oss01.local_UUID 3
1 UP ost OSS OSS_UUID 3
```

3. On the clients, start the other side of the echo connection, run:

```
$ lctl dl
0 UP osc OSC_xfer01.local_ost_oss01.local_ECHO_client \
6bc9b_ECHO_client_2a8a2cb3dd 5
1 UP echo_client ECHO_client 6bc9b_ECHO_client_2a8a2cb3dd 3
```

4. Verify connectivity from a client, run:

```
$ lctl ping SERVER_NID
```

5. Run the script on the master node, specifying the client names in an environment variable. For example, run:

```
$ client_names_str='xfer01:ECHO_client xfer02:ECHO_client
xfer03:ECHO_client xfer04:ECHO_client xfer05:ECHO_client
xfer06:ECHO_client xfer07:ECHO_client xfer08:ECHO_client
xfer09:ECHO_client xfer10:ECHO_client xfer11:ECHO_client
xfer12:ECHO_client' ./obdfilter-survey
```

6. When aborting, run killall vmstat on clients:

```
pdsh -w (clients) killall vmstat
```

Use `lctl device_list` to verify the `obdfilter/echo_client` instance names. For example, when the script runs, it creates a number of working files and a pair of result files. All files start with the prefix given by `${rslt}`.

```
${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 script iterates over the given numbers of threads and objects performing all specified tests and checking that all test processes completed successfully.

Note that the script does not clean up properly if it is aborted or if it encounters an unrecoverable error. In this case, 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`.

19.2.2.3 Output of the sbdfilter_survey Script

The summary file and stdout contain lines like:

```
ost 8 sz 67108864K rsz 1024 obj8 thr8 write613.54 [ 64.00, 82.00]
```

Where:

Variable	Description
ost8	Total number of OSTs under test
sz 67108864K	Total amount of data read or written (in KB)
rsz 1024	Record size (size of each echo_client I/O)
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
[64, 82.00]	Minimum and maximum instantaneous bandwidths seen on any individual OST

Note – Although the numbers of threads and objects are specified per-OST in the customization section of the script, results are reported aggregated over all OSTs.

19.2.2.4 Visualizing Results

It is useful to import the summary data (its fixed width) into Excel (or any graphing package) and graph the bandwidth against 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 inputs/outputs in flight.

It is also useful to record average disk I/O sizes during each test. These numbers help find pathologies in the system when the filesystem block allocator or the block device elevator fragment I/O requests.

The `obparse.pl` script (included) is an example of processing the output files to a `.csv` format.

19.2.3 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 filesystem, 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
```

19.3 PIOS Test Tool

The PIOS test tool is a parallel I/O simulator for Linux and Solaris. PIOS generates I/O on filesystems, 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: <ftp://ftp.clusterfs.com/pub/lustre/pios>

19.3.1 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]
```

19.3.2 PIOS I/O Modes

There are several supported PIOS I/O modes:

POSIX IO:

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 IO:

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 filesystem on an OSS.

COW IO:

This mode corresponds to the copy overwrite operation where filesystem 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.

19.3.3 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 filesystem 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
<code>--chunknoise = N</code>	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.
<code>--chunksize = N[,N2,N3...]</code>	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 <code>--chunksize_low</code> . Note that each thread allocates a buffer of size <code>chunksize + chunknoise</code> for use during the run.
<code>--chunksize_low=L</code> <code>--chunksize_high=H</code> <code>--chunksize_incr=F</code>	Performs a sequence of operations starting with a chunksize of L, increasing it by F% each time until chunksize exceeds H.
<code>--cleanup</code>	Removes files that were created during the run. If there is an encounter for existing files, they are over-written.
<code>--directio</code> <code>--posixio</code> <code>--cowio</code>	One of these arguments must be passed to indicate if DIRECT I/O, POSIX I/O or COW I/O is used.
<code>--offset=O[,O2,O3...]</code>	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.
<code>--offset_low=OL</code> <code>--offset_high=OH</code> <code>--offset_inc=PH</code>	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.
<code>--prerun="pre-command"</code>	Before each run, executes the pre-command as a shell command through the <code>system(3)</code> 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.
<code>--postrun="post-command"</code>	After each run, executes the post-command as a shell command through the <code>system(3)</code> 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.
<code>--regioncount=N[,N2,N3...]</code>	PIOS writes to N regions in a single file or block device or to N files.
<code>--regioncount_low=RL</code> <code>--regioncount_high=RH</code> <code>--regioncount_inc=P</code>	Generate runs with a range of region counts starting at RL, increasing P% until the thread count exceeds RH. Each of these arguments is exclusive with the regioncount argument.

<code>--regionnoise=k</code>	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 <i>k</i> regions ahead. The run will complete when all regions are fully written or read. This merely introduces a randomization of the ordering.
<code>--regionsize=S[,S2,S3...]</code>	The argument is a byte specifier or a list of byte specifiers. During the run(s), write <i>S</i> bytes to each region.
<code>--regionsize_low=RL</code> <code>--regionsize_high=RH</code> <code>--regionsize_inc=P</code>	The arguments are byte specifiers. Generate runs with a range of region sizes starting at <i>TL</i> , increasing <i>P</i> % until the region size exceeds <i>RH</i> . Each argument is exclusive with the <code>regionsize</code> argument.
<code>--threadcount=T[,T2,T3...]</code>	PIOS runs with <i>T</i> threads performing I/O. A sequence of values may be given.
<code>--threadcount_low=TL</code> <code>--threadcount_high=TH</code> <code>--threadcount_inc=TP</code>	Generate runs with a range of thread counts starting at <i>TL</i> , increasing <i>TP</i> % until the thread count exceeds <i>TH</i> . Each of these arguments is exclusive with the <code>threadcount</code> argument.
<code>--threaddelay=ms</code>	A random amount of noise not exceeding <i>ms</i> 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.
<code>--fpp</code>	Where threads write to files: <ul style="list-style-type: none"> • 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.
<code>--verify-V=timestamp</code> <code>[,timestamp2,timestamp3]</code> <code> --verify -V</code>	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.

19.3.4 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
```

19.4 LNET Self-Test

LNET self-test helps site administrators confirm that Lustre Networking (LNET) has been properly installed and configured. The self-test also confirms that LNET and the network software and hardware underlying it are performing according to expectations.

19.4.1 Introduction to LNET Self-Test

This section describes basic concepts of LNET self-test, modules, utilities and a sample script.

Every LNET self-test runs in the context of a *session*. A node can be associated with only one session at a time to ensure the session has exclusive use of the nodes on which it is running. A session is created, controlled and monitored from a single node; this 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 simply 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.

19.4.1.1 Modules

To run LNET self-test, load these nodes: `libcfs`, `lnet`, `lnet_selftest` and any one of `klnds` (i.e. `ksocklnd`, `ko2iblnd`...). There are two types of nodes for LNET self-test: console node and test nodes. Both node types require all previously-specified modules to be loaded. (The userspace test node does not require these modules.)

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.

19.4.1.2 Utilities

LNET self-test has 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 use `lstclient` to join a self-test session:

```
lstclient -sesid CONSOLE_NID group NAME
```

19.4.1.3 Sample Script

Below is a sample 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.

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).

19.4.2 LNET Self-Test Concepts

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

19.4.2.1 Session

In the context of LNET self-test, a session is a 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. (The only exception is `show_session` command which is used to query the session). A user can only operate nodes in his own session. Session context in all test nodes will be destroyed if the session ends. Several session commands (`new_session`, `end_session`, `show_session`) are introduced later in this topic.

19.4.2.2 Console

A console can be any node, whether in or out of the session. A console is the user interface of the self-test system. All self-test commands are entered from a console. A user can control and monitor status of the entire testing cluster (session) from the console. A console node is exclusive, meaning that a user cannot control two different sessions on one node.

19.4.2.3 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.

19.4.2.4 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 the test type, source group, target group, or concurrency, etc.

19.4.2.5 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, it should not exist individually. Users can control a test batch (run, stop), they cannot control individual tests.

19.4.3 LNET Self-Test Commands

LNET self-test commands are issued using the `lst` command line utility. This utility takes a number of command line arguments. The first argument is the command name, while subsequent arguments are command-specific.

19.4.3.1 Session

Process Environment

`lst` uses the `LST_SESSION` environment 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 will end 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 a node has active session, an attempt to create a new session fails unless this flag is specified. Similarly, if a session attempts to add a node that is already owned by another session, this flag allows the 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 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
```

19.4.3.2 Group

add_group NAME NIDLIST [NIDLIST...]

Adds a list of test nodes to a group (and creates the group if it does not already exist).

NAME The name of the group.

NIDLIST 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 IDLIST]

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 IDLIST 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 current session if no group is specified.

- NAME The name of the group.
- active Lists the active node NIDs.
- busy Lists the busy node NIDs.

-down Lists the down node NIDs.

-unknown Lists unknown node NIDs.

-all Lists all node NIDs.

```
$ 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 a session. If the group is referred by any test, the operation fails. If nodes in the group are referred only by this group, they are kicked out from current session; otherwise, they are still in the current session.

```
$ lst del_group clients
```

Userland client (lstclient -sesid NID -group NAME)

A user can run userland self-test client by `lstclient`. `lstclient` should be executed after creating a session on the console. There are only two 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
```

19.4.3.3 Batch and Test

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 a 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)
```

Now we have only two test types:

-ping 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 validation check, either full or simple. The default is no-check.
This is 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
// 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 are listed. If an option is use, only the specified test is 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
```

19.4.3.4 Other Commands

This section lists other LNET self-test commands.

*ping [session] [group NAME] [nodes IDLIST] [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 IDLIST | Pings all of the 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 <u>only</u> 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 | IDs [GROUP | IDs]*

The collection performance and RPC statistics from a specified group. Currently, we can only get LNET performance statistics. In the future, more statistics will be supported. 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 value is 5 seconds.
-delay #	The interval of the statistics.

```
$ lst run bulkperf
$ lst stat clients
[LNet Rates of clients]
[W] Avg: 1108 RPC/s Min: 1060 RPC/s Max: 1155 RPC/s
10
[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
```

show_error [session] [GROUP] | [IDs] ...

Lists the number of failed RPCs on the test nodes.

-session Only 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
```


LustreProc

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

- [Introduction](#)
- [Lustre I/O Tunables](#)
- [Debug Support](#)

20.1 Introduction

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

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

20.1.1 /proc Entries for Lustre

This section includes /proc entries for Lustre.

20.1.1.1 Finding Lustre

By using the proc files on the MGS, you can see the following:

- All known filesystems

```
# cat /proc/fs/lustre/mgs/MGS/filesystems
spfs
lustre
```

- The server names participating in a filesystem (for each filesystem 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
```

20.1.1.2 Lustre Timeouts/ Debugging

/proc/sys/lustre/timeout

This is the time period that a client waits for a server to complete an RPC (default 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."

/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.

20.1.1.3 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
```

Fields are explained below:

Field	Description
refs	A reference count (principally used for debugging)
state	Only valid to refer to routers. Possible values: <ul style="list-style-type: none">• ~ 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 than `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 current queue health on this node.

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
```

20.1.1.4 Free Space Distribution

The free-space stripe weighting is set to give a priority of "0" to the free space (versus trying to place the stripes "widely" -- nicely distributed across OSSs and OSTs to maximize network balancing).

You can adjust this priority via the proc file:

```
$ 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% just means that OSS distribution does not count in the weighting, but the stripe assignment is still done via weighting. If OST2 has twice as much free space as OST1, it will be twice as likely to be used, but it is NOT guaranteed to be used.

Also note that free-space stripe weighting does not activate until two OSTs are imbalanced by more than 20%. Until then, a faster round-robin stripe allocator is used. (The new round-robin order also maximizes network balancing.)

20.2 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).

20.2.1 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 that the OSC will issue at a time 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.

The value for `max_dirty_mb` is recommended to be `4 * max_pages_per_rpc * max_rpcs_in_flight` in order to maximize performance.

Note – The *<object name>* varies depending on the specific Lustre configuration. For *<object name>* examples, refer to the sample command output.

20.2.2 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
pages per rpc  rpcs  %   cum%   |   write
1:             0    0   0       |   0    0   0

      read
rpcs in flight  rpcs  %   cum%   |   write
0:             0    0   0       |   0    0   0

      read
offset         rpcs  %   cum%   |   write
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.

20.2.3 Client Read-Write Offset Survey

The `rw_offset_stats` maintains statistics for the 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.

Example:

```
# cat /proc/fs/lustre/llite/lustre-f57dee00/rw_offset_stats
snapshot_time: 1155748884.591028 (secs.usecs)
R/W  PID  RANGE  STARTRANGEEND  SMALLESTEXTENTLARGESTEXTENT  OFFSET
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, <code>Smallest-Extent</code> 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 <code>rw_offset_stats</code> file can be cleared by writing to it:

```
echo > /proc/fs/lustre/llite/lustre-f57dee00/rw_offset_stats
```

20.2.4 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/spfs-c45f9c00/extent_stats
snapshot_time:      1174868361.372513 (secs.usecs)
      read          |          write
extents      calls % cum% |      calls % cum%
OK - 4K :      0   0  0   |      0   0  0
```

The file can be cleared by issuing the following command:

```
$ echo > cat /proc/fs/lustre/llite/spfs-c45f9c00/extent_stats
```

Per-Process Client I/O Statistics

The `extent_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/spfs-c45f9c00/extent_stats_per_process
snapshot_time:      1174868461.566452 (secs.usecs)
      read          |          write
extents      calls % cum% |      calls % cum%
```

20.2.5 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 <code>rpc_stats</code> .
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 filesystem for a single RPC.

20.2.6 Mechanics of Lustre Readahead

Readahead is a method of reading part of a file's contents into memory with the expectation that a process working with the file will soon want the data. When readahead works well, a data-consuming process finds that the information it needs is available when it asks, and waiting for disk I/O is not necessary.

Lustre 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. Setting this tunable to 0 disables readahead. 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().

20.2.7 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 filesystem 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	found	grps	cr	\	merge	tail	broken
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 filesystem is pretty fragmented or mballocc is doing something wrong in the group selection part.

20.2.8 mballocc3 Tunables

mballocc3 is included in Lustre version 1.6.1 and later. mballocc3¹ was built on top of mballocc2, 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 mballocc3 tunables are currently available:

Parameter	Description
stats	Enables/disables the collection of statistics. Collected statistics can be found in <code>/proc/fs/ldiskfs2/<dev>/mb_history</code> .
max_to_scan	Maximum number of free chunks that mballocc finds before a final decision to avoid livelock.
min_to_scan	Minimum number of free chunks that mballocc 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 $N \geq \text{order2_req}$), 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.

¹mballocc3 is enabled, by default.

The following tunables, providing more control over allocation policy, will be available in the next version:

Parameter	Description
stats	Enables/disables the collection of statistics. Collected statistics can be found in <code>/proc/fs/ldiskfs2/<dev>/mb_history</code> .
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 $N \geq \text{order2_req}$), a very fast search via buddy structures is used.
small_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.
large_req	
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.

20.2.9 Locking

*/proc/fs/lustre/ldlm/ldlm/namespaces/<OSC name | MDCname>
/lru_size*

This variable determines how many locks can be queued up on the client in an LRU queue. The default value of LRU size is 100. Increasing this on a large number of client nodes is not recommended, though servers have been tested with up to 150,000 total locks (**num_clients** * **lru_size**). Increasing it for a small number of clients (for example, login nodes with a large working set of files due to interactive use) can speed up Lustre dramatically. Recommended values are in the neighborhood of 2500 MDC locks and 1000 locks per OSC.

The following command can be used to clear the LRU on a single client, and as a result flush client cache, without changing the LRU size value:

```
$ echo clear > /proc/fs/lustre/ldlm/ldlm/namespaces/<OSC \  
name|MDC name>/lru_size
```

If you shrink the LRU size below the number of existing unused locks, the locks are canceled immediately. Use `echo clear` to cancel all locks without changing the value.

20.3 Debug Support

/proc/sys/lnet/debug

Lustre generates a detailed log of all its operations to aid in debugging by default. This can affect the performance or speed you achieve with Lustre. Therefore, it is useful to reduce this overhead by turning down the debug level. 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 portals.debug
portals.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 portals.debug=0
portals.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 for subsystems (see S_* definitions).

/proc/sys/lnet/debug_path

This indicates the location where debugging symbols should be stored for gdb. The default is set to `/r/tmp/lustre-log-localhost.localdomain`.

These values can also be set via `sysctl -w lnet.debug={value}`

Note – The above entries only exist when Lustre has already been loaded.

Lustre uses the set debug level after it is loaded on a particular node. You can set the debug level by adding the following to the node entry config shell script:

```
--ptldebug <level>
```

20.3.1 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-
d24265c91a3d/stats
/proc/fs/lustre/mds/lustre-MDT0000/exports/08ac6584-6c4a-3536-2c6d-
b36cf9cbdaa0/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 <code>ost_destroy</code> 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
<code>req_waitempty</code>	Amount of time a request waited in the queue before being handled by an available server thread.
<code>req_qdepth</code>	Number of requests waiting to be handled in the queue for this service.
<code>req_active</code>	Number of requests currently being handled.
<code>reqbuf_avail</code>	Number of unsolicited <code>lnet</code> request buffers for this service.

Some service-specific events of interest are:

Parameter	Description
<code>ldlm_enqueue</code>	Time it takes to enqueue a lock (this includes file open on the MDS)
<code>mds_reint</code>	Time it takes to process an MDS modification record (includes create, mkdir, unlink, rename and setattr)

Lustre Tuning

This chapter contains information to tune Lustre for better performance and includes the following sections:

- [Module Options](#)
- [LNET Tunables](#)
- [Options to Format MDT and OST Filesystems](#)
- [Network Tuning](#)
- [DDN Tuning](#)
- [Large-Scale Tuning for Cray XT and Equivalents](#)

21.1 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`).

21.1.0.1 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.

Note – Consider memory consumption when increasing the thread pool size.

21.1.1 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.

21.2 LNET Tunables

This section describes LNET tunables.

21.2.0.1 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.

21.2.0.2 irq_affinity

By default, this parameter is ON. In the normal case on an SMP system, we would like network traffic to remain local to a single CPU. This helps to keep the processor cache warm, and minimizes the impact of context switches. This is especially helpful when an SMP system has more than one network interface, and ideal when the number of interfaces equals the number of CPUs.

If you have an SMP platform with a single fast interface such as 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.

21.3 Options to Format MDT and OST Filesystems

The backing filesystems on an MDT and OSTs are independent of one another, so the formatting parameters for them should not be same. The size of the MDT backing filesystem depends solely on how many inodes you want in the entire Lustre filesystem. This is not related to the size of the aggregate OST space.

21.3.1 Planning for Inodes

Each time you create a file on a Lustre filesystem, 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 filesystems, 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 filesystem is formatted. If there is a shortage of inodes or space on the OSTs, it is possible to add OSTs to the filesystem.

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 filesystem installations. Many Lustre filesystems have average file sizes over 1 MB per object.

21.3.2 Calculating MDT Size

When calculating the MDT size, the only important factor is the average size of files to be stored in the filesystem. If the average file size is, for example, 5 MB and you have 100 TB of usable OST space, then you need at least $(100 \text{ TB} * 1024 \text{ GB/TB} * 1024 \text{ MB/GB} / 5 \text{ MB/inode}) = 20 \text{ million inodes}$. 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.

21.3.3 Overriding Default Formatting Options

To override the default formatting options for any of the Lustre backing filesystems, use the `--mkfsoptions='backing fs options'` argument to `mkfs.lustre`, to pass formatting options to the backing `mkfs`. For all options to format backing `ext3` and `ldiskfs` filesystems, see the `mke2fs(8)` man page; this section only discusses several Lustre-specific options.

21.3.3.1 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 filesystem space). 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 filesystem 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`).

21.3.3.2 Inode Size for MDT

Lustre uses "large" inodes on backing filesystems 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` filesystem), 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 filesystem. The inode ratio must always be larger than the inode size.

21.3.3.3 Number of Inodes for OST

For OST filesystems, it is normally advantageous to take local filesystem 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 filesystem (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:

$$\text{num_ost_inodes} = 4 * \text{<num_mds_inodes>} * \text{<default_stripe_count>} / \text{<number_osts>}$$

To specify the number of inodes on OST filesystems, 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 filesystems 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 filesystems, see [Formatting](#).

21.4 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'

21.5 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

21.5.1 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 $8 * 1000 * 1 \text{ MB} = 8 \text{ GB}$ 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.

21.5.2 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
```

21.5.3 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.

21.5.4 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.

21.5.5 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 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 (Mbytes)	Block Size	Tiers	Tier	list

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							

21.6 Large-Scale Tuning for Cray XT and Equivalents

This section only applies to Cray XT3 Catamount nodes, and explains parameters used with the kptlnd module. If it does not apply to your setup, ignore it.

21.6.1 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	<p>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.</p> <p>Max nodes > (Total # Nodes) * (max_procs_per_node)</p> <p>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.</p>
max_procs_per_node	<p>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, then LNET removes the oldest one to make space for the new one.</p>

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	<p>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.</p>
Credits	<p>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.</p>

21.7 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/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.

Lustre Troubleshooting Tips

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

- [Lustre Error Messages and Logs](#)
- [Lustre Performance Tips](#)

22.1 Lustre Error Messages and Logs

To effectively debug Lustre, you need to review the Lustre error messages and logs.

22.1.1 Lustre Error Messages

As Lustre code runs on the kernel, single-digit error codes display to the application; these error codes are an indication of the problem. Refer to the kernel console log (`dmesg`) for all recent kernel messages from that node. On the node, `/var/log/messages` holds a log of all messages for at least the past day.

22.1.2 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 these logs on each of the nodes involved, run

```
$ lctl dk <filename>
```

22.2 Lustre Performance Tips

This section describes various tips to improve Lustre performance.

22.2.1 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.

22.2.2 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 filesystems that use socklnd (TCP, Ethernet) as interconnect, there is also additional CPU overhead because the client cannot receive data without copying it from the network buffers. In the write case, the client CAN send data without the additional data copy. This means that the client is more likely to become CPU-bound during reads than writes.

22.2.3 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 filesystem 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 filesystem returns this error if it is unable to read from the storage device.

Suggested Action

If the reported error is -2, you can consider checking in `/lost+found` on your raw OST device, to see if the missing object is there. However, it is likely that this object is lost forever, and that the file that references the object is now partially or completely lost. Restore this file from backup, or salvage what you can and delete it.

If the reported error is anything else, then you should immediately inspect this server for storage problems.

22.2.4 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 filesystem corruption. This is a normal behavior. The status in `/proc/fs/lustre/healthcheck` also shows "not healthy" on the affected nodes.

To recover from this problem, you must restart Lustre services using these filesystems. There is no other way to know that the I/O made to disk, and the state of the cache may be inconsistent with what is on disk.

22.2.5 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. On MDS and client nodes, run:

```
# lctl dl
```

2. Deactivate the OST, run:

```
# lctl --device N deactivate
```

Note that N will be different for the MDS and clients.

Note – If the OST later becomes available it needs to be reactivated, run:

```
# lctl --device N 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 filesystem.

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 the `munlink` command 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 striping pattern (which includes the index of the missing OST). Run:

```
# lfs getstripe -v {filename}
```

7. Use this computation to determine which offsets in the file are affected:

$[(C*N + X)*S, (C*N + X)*S + S - 1]$, $N = \{ 0, 1, 2, \dots \}$

where:

C = stripe count

S = stripe size

X = index of bad OST for this file

For example, for a file with 2 stripes, stripe size = 1M, bad OST is at index 0, then you would have holes in your file at:

$[(2*N + 0)*1M, (2*N + 0)*1M + 1M - 1]$, $N = \{ 0, 1, 2, \dots \}$

If the filesystem 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 filesystem 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 filesystem with a missing OST.

22.2.6 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 `lctl`.

With `mkfs.lustre`

While you are using the `mkfs` command and creating the filesystem, 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
```


22.2.7 Default Striping

These are the default striping settings:

```
lov.stripesize=<bytes>
lov.stripecount=<count>
lov.stripeoffset=<offset>
```

To change the default striping information.

- On the MGS:

```
$ lctl conf_param testfs-MDT0000.lov.stripesize=4M
```

- On the MDT and clients:

```
$ mdt/cli> cat /proc/fs/lustre/lov/testfs-{mdt|cli}lov/stripe*
```

22.2.8 Erasing a Filesystem

If you want to erase a filesystem, run this command on your targets:

```
$ "mkfs.lustre -reformat"
```

If you are using a separate MGS and want to keep other filesystems defined on that MGS, then set the `writeconf` flag on the MDT for that filesystem. 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 filesystem, run:**

```
$ umount /mnt/lustre
```

2. **Erase the filesystem and, presumably, replace it with another filesystem, 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 filesystem. Nothing needs to be done with old disks that will not be part of the new filesystem, just do not mount them.

22.2.9 Reclaiming Reserved Disk Space

All current Lustre installations run the ext3 filesystem internally on service nodes. By default, the ext3 reserves 5% of the disk space for the root user. In order to reclaim this space, run the following command on your OSSs:

```
tune2fs [-m reserved_blocks_percent] [device]
```

You do not need to shut down Lustre before running this command or restart it afterwards.

22.2.10 Considerations in Connecting a SAN with Lustre

Depending on your cluster size and workload, you may want to connect a SAN with Lustre. Before making this connection, consider the following:

- In many SAN filesystems 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.
- Filesystems 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 filesystem for many reasons like improper filesystem, network, or other kernel software, bad cabling, bad memory, and so on. The risk increases with increase in the number of clients directly accessing the storage.

22.2.11 Handling/Debugging "Bind: Address already in use" Error

During startup, Lustre may report a `bind: Address already in use` error and reject to start the operation. This is caused by a portmap service (often NFS locking) which starts before Lustre and binds to the default port 988.

Unfortunately, you cannot set `sunrpc` 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 lneth accept_port=988
```

- Add `modprobe ptlrpc` to your system startup scripts before the service that uses `sunrpc`. This causes Lustre to bind to port 988 and `sunrpc` to select a different port.

Note – You can also use the `sysctl` command to mitigate the NFS client from grabbing the Lustre service port. However, this is a partial workaround as other user-space RPC servers still have the ability to grab the port.

22.2.12 Replacing An Existing OST or MDS

The OST filesystem is simply a normal ext3 filesystem. 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 filesystems.

For example:

```
mount -t ext3 /dev/old /mnt/ost_old
mount -t ext3 /dev/new /mnt/ost_new
rsync -aSv /mnt/ost_old/ /mnt/ost_new
# note trailing slash on ost_old/
```

- If you are unable to connect both sets of disk to the same computer, use `rsync` to copy over the network using `rsh` (or `ssh` with `-e ssh`):

```
rsync -aSvz /mnt/ost_old/ new_ost_node:/mnt/ost_new
```

- Use the same procedure for the MDS, with one additional step:

```
cd /mnt/mds_old; getfattr -R -e base64 -d . > /tmp/mdsea; \
<copy all MDS files as above>; cd /mnt/mds_new; setfattr \
--restore=/tmp/mdsea
```

22.2.13 Handling/Debugging Error "- 28"

Linux error -28 is -ENOSPC and indicates that the filesystem has run out of space. You need to create larger filesystems 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 filesystem 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 textual description.

22.2.14 Triggering Watchdog for PID NNN

In some cases, a server node triggers a watchdog timer and this causes a process stack to be dumped to the console along with a Lustre kernel debug log being dumped into `/tmp` (by default). The presence of a watchdog timer does NOT mean that the thread OOPSed, but rather that it is taking longer time than expected to complete a given operation. In some cases, this situation is expected.

For example, if a RAID rebuild is really slowing down I/O on an OST, it might trigger watchdog timers to trip. But another message follows shortly thereafter, indicating that the thread in question has completed processing (after some number of seconds). Generally, this indicates a transient problem. In other cases, it may legitimately signal that a thread is stuck because of a software error (lock inversion, for example).

```
Lustre: 0:0:(watchdog.c:122:lcw_cb())
```

The above message indicates that the watchdog is active for pid 933:

It was inactive for 100000ms:

```
Lustre: 0:0:(linux-debug.c:132:portals_debug_dumpstack())
```

Showing stack for process:

```
933 ll_ost_25      D F896071A      0   933      1   934   932 (L-TLB)
f6d87c60 00000046 00000000 f896071a f8def7cc 00002710 00001822 2da48cae
0008cf1a f6d7c220 f6d7c3d0 f6d86000 f3529648 f6d87cc4 f3529640 f8961d3d
00000010 f6d87c9c ca65a13c 00001fff 00000001 00000001 00000000 00000001
```

Call trace:

```
filter_do_bio+0x3dd/0xb90 [obdfilter]
default_wake_function+0x0/0x20
filter_direct_io+0x2fb/0x990 [obdfilter]
filter_preprw_read+0x5c5/0xe00 [obdfilter]
lustre_swab_niobuf_remote+0x0/0x30 [ptlrpc]
ost_brw_read+0x18df/0x2400 [ost]
ost_handle+0x14c2/0x42d0 [ost]
ptlrpc_server_handle_request+0x870/0x10b0 [ptlrpc]
ptlrpc_main+0x42e/0x7c0 [ptlrpc]
```

22.2.15 Handling Timeouts on Initial Lustre Setup

If you come across timeouts or hangs on the initial setup of your Lustre system, verify that name resolution for servers and clients is working correctly. Some distributions configure `/etc/hosts` so the name of the local machine (as reported by the 'hostname' command) is mapped to local host (127.0.0.1) instead of a proper IP address.

This might produce this error:

```
LustreError:(ldlm_handle_cancel()) received cancel for unknown lock cookie
0xe74021a4b41b954e from nid 0x7f000001 (0:127.0.0.1)
```

22.2.16 Handling/Debugging "LustreError: xxx went back in time"

Each time Lustre changes the state of the disk filesystem, it records a unique transaction number. Occasionally, when committing these transactions to the disk, the last committed transaction number displays to other nodes in the cluster to assist the recovery. Therefore, the promised transactions remain absolutely safe on the disappeared disk.

This situation arises when:

- You are using a disk device that claims to have data written to disk before it actually does, as in case of a device with a large cache. If that disk device crashes or loses power in a way that causes the loss of the cache, there can be a loss of transactions that you believe are committed. This is a very serious event, and you should run `e2fsck` against that storage before restarting Lustre.
- As per the Lustre requirement, the shared storage used for failover is completely cache-coherent. This ensures that if one server takes over for another, it sees the most up-to-date and accurate copy of the data. In case of the failover of the server, if the shared storage does not provide cache coherency between all of its ports, then Lustre can produce an error.

If you know the exact reason for the error, then it is safe to proceed with no further action. If you do not know the reason, then this is a serious issue and you should explore it with your disk vendor.

If the error occurs during failover, examine your disk cache settings. If it occurs after a restart without failover, try to determine how the disk can report that a write succeeded, then lose the Data Device corruption or Disk Errors.

22.2.17 Lustre Error: "Slow Start_Page_Write"

The `slow_start_page_write` message appears when the operation takes an extremely long time to allocate a batch of memory pages. Use these pages to receive network traffic first, and then write to disk.

22.2.18 Drawbacks in Doing Multi-client O_APPEND Writes

It is possible to do multi-client O_APPEND writes to a single file, but there are few drawbacks that may make this a sub-optimal solution. These drawbacks are:

- Each client needs to take an EOF lock on all the OSTs, as it is difficult to know which OST holds the end of the file until you check all the OSTs. As all the clients are using the same O_APPEND, there is significant locking overhead.
- The second client cannot get all locks until the end of the writing of the first client, as the taking serializes all writes from the clients.
- To avoid deadlocks, the taking of these locks occurs in a known, consistent order. As a client cannot know which OST holds the next piece of the file until the client has locks on all OSTs, there is a need of these locks in case of a striped file.

22.2.19 Slowdown Occurs During Lustre Startup

When Lustre starts, the Lustre filesystem 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 filesystem 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 filesystem now reads data from the cache.

22.2.20 Log Message ‘Out of Memory’ on OST

When planning the hardware for an OSS node, you must 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 "kmallocc 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](#).

Lustre Debugging

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

- [Lustre Debug Messages](#)
- [Tools for Lustre Debugging](#)
- [Using Strace for Troubleshooting](#)
- [Looking at Disk Content](#)
- [Ptlrpc Request History](#)
- [LWT Tracing](#)

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, ldln, lov
- Debug Types:
 - otrace: Entry/Exit markers
 - odlmltrace: Locking related information
 - oinode
 - osuper
 - oext2: Anything from ext2_debug
 - omalloc: Print malloc or free information
 - ocache: Cache related information
 - oinfo: General information
 - oioclt: IOCTL related information
 - oblocks: Ext2 block allocation information
 - onet: Networking
 - owarning
 - obuffs
 - oother
 - odentry
 - oportals: Entry/ Exit markers
 - opage: Bulk page handling
 - oerror: Error messages
 - oemerg
 - orpctrace: For distributed debugging
 - oha: Information related to failover and recovery

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.

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 substantial amount of debugging information (MBs or more) during the first insertion of the kernel module. When this buffer fills up, it wraps around and throws away the oldest information. Lustre offers added debug messages that can be written out to this kernel log.
- **Debug daemon:** The debug daemon provides facility for unlimited logging of the CDEBUG logs in Lustre.
- **/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 provides filtering the kernel to extract the useful information.
- **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 some kernels, enabled sysrq "c" produces a crashdump. Lustre enhances this crash dump with a log dump (the last 64K of log) to the console.
- **debugfs:** Interactive ext2 filesystem 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).

23.2.1 Lustre Debug Daemon

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 Debug Daemon Commands

`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 path 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:

```
sort -k4 -t: > {newfile}
```

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.

EXAMPLE

This is an example using `debug_daemon` 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.).

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

To completely turn off Lustre debugging:

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

To enable full debugging:

```
sysctl -w lnet.debug=net
```

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 logging of messages related to network communications, and leave debug flags alone.

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 special 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
```

- **LBUG:** This is a panic style assertion in the kernel which causes Lustre to dump its circular log to the file `/tmp/lustre-log` from where it can be retrieved after a reboot.
- **CDEBUG:** CDEBUG is the basic and most commonly used debug macro, and takes just one more argument than a standard `printf` - the debug type. This message adds to the debug log with the debug mask set accordingly. Later, when a troubleshooter retrieves the log, they can also filter based on this type.
`CDEBUG(D_INFO, "This is my debug message: the number is %d\n", number);`
- **CERROR:** 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:** ENTRY and EXIT take no arguments and simply add messages to aid in call tracing. You should attempt to cover all exit conditions when using these macros, to avoid confusion when the debug log reports that a function was entered but never exited.
- **LDLM_DEBUG and LDLM_DEBUG_NOLOCK:** These macros are to be used when tracing MDS and VFS operations for locking. The purpose of these macros is to build a thin trace that shows the protocol exchanges between nodes.

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`.

23.3 Using Strace for Troubleshooting

The operating system makes `strace` (system call trace utility) available. Use `strace` to trace a system call, intercept system calls made by a process, and record the system call name, arguments, and return values. This is a very useful tool, especially when you try to troubleshoot a failed system call.

To invoke `strace` for a system call:

```
$strace <system-call> <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 <system-call> <args>
```

To redirect the `strace` output to a file (to review at a later time):

```
$strace -o <filename> <system-call> <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 extended attributes of the file inode stores the information about objects that make a file. This contains a list of all the object IDs and their locations (that is, which OST device stores them). Lustre uses the `lfind` tool (in `lustre/utls`) to obtain this information for a given filename. Lustre uses the corresponding `lstripe` tool to modify the extended attributes of a file.

The `lfind` 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 filesystem:

```
$lfind /mnt/lustre/frog
$
OBDs:
  0 : OSC_localhost_UUID
  1 : OSC_localhost_2_UUID
  2 : OSC_localhost_3_UUID
obdidx objid
  0   17
  1    4
```

The operating system provides the `debugfs` tool to interactively debug an ext2 filesystem. It helps to either check the status or modify some information in the filesystem (maybe to correct the filesystem state).

Lustre stores all objects that belong to a file in an underlying extN filesystem on the OSTs. The filesystem uses the object IDs as the filenames. Once the object IDs are known, `debugfs` uses them to obtain the attributes of all objects from the different OSTs. This is a sample run for the `/mnt/lustre/frog` file used in the example above:

```
$debugfs -c /tmp/ost1
debugfs: cd 0
debugfs: cd R /* for regular files*/
debugfs: cd d<objid % 32>
debugfs: stat <objid> /* for getattr on object*/
debugfs: quit
```

For example, if an object ID is 36, use these steps:

```
$debugfs /tmp/ost1
debugfs: cd 0
debugfs: cd R                      /* for regular files */
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 Tcpcmdump

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 `tcpcmdump` 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 tcpcmdump
```

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>
```

Where:

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	<ul style="list-style-type: none">• 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 LWT Tracing

LWT is a lightweight tracing facility. It prints fixed size requests into a buffer and is faster than LDEBUG.

The records dumped by LWT contain:

- Current CPU
- Process counter
- Pointer to file
- Pointer to line in the file
- 4 void * pointers

The `lctl` command dumps the logs to files. This has been very successful in debugging difficult problems.

PART IV Lustre for Users

This part includes chapters on Lustre striping and I/O options, security and operating tips.

Free Space and Quotas

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

- [Querying Filesystem Space](#)
- [Using Quota](#)

24.1 Querying Filesystem Space

The `lfs df` command is used to determine available disk space on a filesystem. It displays the amount of available disk space on the mounted Lustre filesystem and shows space consumption per OST. If multiple Lustre filesystems 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.

Examples

```
[lin-clil] $ 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-clil] $ 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-clil] $ 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 Quota

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 filesystem 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 filesystem.

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 filesystem only when you actually write the data to these blocks.

Striping and I/O Options

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

- [File Striping](#)
- [Displaying Files and Directories with `lfs getstripe`](#)
- [lfs setstripe – Setting Striping Patterns](#)
- [Free Space Management](#)
- [Performing Direct I/O](#)
- [Other I/O Options](#)
- [Striping Using `llapi`](#)

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 creation/stripping 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 filesystem, 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 Striping Patterns

Use the `lfs setstripe` command to create new files with a specific stripe configuration.

```
lfs setstripe <filename|dirname> [--size|-s stripe-size]<k|m|g> \
[--index|-i stripe_index] [--count|-c stripe_count]
```

Stripe-Size

If you pass a stripe-size of 0, the filesystem's default stripe size is used. Otherwise, the stripe-size must be a multiple of 64 KB. It can be specified with k, m or g (in KB, MB or GB respectively).

Starting-OST

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 filesystem'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 want to adjust only the stripe-count and keep the other parameters at their default settings, use this syntax:

```
lfs setstripe -c <stripe_count> <file>
```

25.3.1 Changing Striping for a Subdirectory

For a directory, the `lfs setstripe` command sets a default striping configuration for files created within 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 filesystem default for the new file.

To change the striping pattern for a sub-directory, create a directory with desired striping pattern as described above. Sub-directories inherit the striping pattern of the parent directory.

Note – Striping on directories only affects NEW files and NEW sub-directories created within them.

25.3.2 Using a Specific Striping Pattern for a Single File

For a single file:

```
lfs setstripe creates a file with a given stripe pattern
lfs setstripe fails if the file already exists
```

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 filesystem 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 Filesystem 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 and striping using `ioctl`.

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 filesystem does NOT do any persistent checksumming, so it does not detect corruption of data in the OST filesystem.

The checksumming feature must be enabled on individual client nodes, and it is currently disabled by default. 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:

```
echo 1 > /proc/fs/lustre/llite/{fsname}/checksum_pages
```

To disable checksums on a client:

```
echo 0 > /proc/fs/lustre/llite/{fsname}/checksum_pages
```

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 filesystem */

    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 (! strcmp(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);
    printf("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/fctest/lustredemo
rm -f /mnt/lustre/fctest/lustre_dummy
cp lustredemo /mnt/lustre/fctest/

```

Lustre Security

This chapter describes Lustre security and includes the following section:

- [Using ACLs](#)

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 filesystem 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, we recommend the SuSE Labs article, “Posix Access Control Lists on Linux” found here:

<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.

26.1.2 Lustre ACLs

Lustre versions 1.4.6 and above support POSIX ACLs. When using a Lustre client at version 1.4.5 or below with an MDS at version 1.4.6, or vice versa, the userspace program generates an “Operation not supported” error during ACL operations.

The MDS needs to be configured to enable ACLs. This can be enabled when creating your configuration with `--mountfsoptions`:

```
$ 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
```

ACLs on the client are enabled at mount time when ACLs are enabled on the MDS. 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 filesystem work exactly like ACLs on any Linux filesystem. 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-head rain
user::rwx
user:chirag:rwx
group::r-x
mask::rwx
other::---
```


Lustre Operating Tips

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

- [Expanding the Filesystem by Adding OSTs](#)
- [A Simple Data Migration Script](#)
- [Adding Multiple SCSI LUNs on Single HBA](#)
- [Failures While Running a Client and an OST on the Same Machine](#)
- [Improving Lustre Metadata Performance While Using Large Directories](#)

27.1 Expanding the Filesystem by Adding OSTs

To add OSTs to existing Lustre filesystems:

- 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 filesystem is quite unbalanced when new empty OSTs are added. New file creations are automatically balanced. If this is a scratch filesystem 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 guard against corruption, the file is checksum'd
# before and after the operation.
# You must supply a temporary directory for the operation.
#

CKSUM=${CKSUM:-md5sum}
MVDIR=$1

if [ $# -ne 1 ]; then
echo "Usage: $0 <dir to copy>"
exit 1
fi

cd $MVDIR

for i in `find . -print`
do
# if directory, skip
if [ -d $i ]; then
echo "dir $i"
else
# Check for write permission
if [ ! -w $i ]; then
echo "No write permission for $i, skipping"
continue
fi

OLDCHK=$(($CKSUM $i | awk '{print $1}'))
NEWNAME=$(mktemp $i.tmp.XXXXXX)
cp $i $NEWNAME
RES=$?
if [ $RES -ne 0 ];then
echo "$i copy error - exiting"
rm -f $NEWNAME
exit 1
fi
```

```

NEWCHK=$(($CKSUM $NEWNAME | awk '{print $1}'))

if [ $OLDCHK != $NEWCHK ]; then
echo "$NEWNAME bad checksum - $i not
moved, \ exiting"
rm -f $NEWNAME
exit 1
else
mv $NEWNAME $i
if [ $RES -ne 0 ];then
echo "$i move error - exiting"
rm -f $NEWNAME
exit 1
fi
fi
fi
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 While Running a Client and an OST on the Same Machine

While running a client and an OST on the same machine, the following failures can occur:

- If the client contains a dirty filesystem in memory and memory pressure, a kernel thread flushes dirty pages to the filesystem, 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 can be improved by:

- Have more RAM on the MDS – On the MDS, more memory translates into bigger caches, thereby increasing the metadata performance.
- Patching 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.

User Utilities (man1)

This chapter describes user utilities and includes the following sections:

- [lfs](#)
- [lfsck](#)
- [Mount](#)
- [Handling Timeouts](#)

28.1 lfs

Use `lfs`, a Lustre client filesystem utility, to display striping information for existing files and to create a file with a specific striping pattern.

28.1.1 Synopsis

```
lfs
lfs find [[]] --atime|-A N] [[]] --mtime|-M N] [[]] --ctime|-C N] \
[--maxdepth|-D N] [--name|-n pattern] [--print0|-P] [--print|-p] \
[--obd|-O <uuid>] <dir|file>
lfs getstripe [--obd|-O <uuid>] [--quiet|-q] [--verbose|-v] \
[--recursive|-r] <dir|file>
lfs setstripe <filename|dirname> <stripe-size> <start-ost>
<stripe-cnt>
lfs setstripe <filename|dirname> [--size|-s stripe-size] \
[--index|-i start-ost] [--count|-c stripe-cnt]
lfs setstripe -d <dirname>
lfs quotachown [-i] <filesystem>
lfs quotacheck [-ug] <filesystem>
```

```
lfs quotaon [-ugf] <filesystem>
lfs quotaoff [-ug] <filesystem>
lfs setquota [-u|-g] <name> <block-softlimit> <block-hardlimit> \
<inode-softlimit> <inode-hardlimit> <filesystem>
lfs setquota -t [ -u | -g ] <block-grace> <inode-grace> <filesystem>
lfs quota [-o obd_uuid] [-u|-g] <name> <filesystem>
lfs check <mds| osts| servers>
lfs df [-i] [-h] [path]
```

Note – In the above example, <filesystem> refers to the mount point of the Lustre filesystem (default is /mnt/lustre).

28.1.2 Description

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

To invoke `lfs` in an interactive mode, run:

```
$ lfs
lfs > help
```

For a complete list of available commands, type `help` at the `lfs` prompt.

Available `lfs` sub-commands are:

setstripe

Creates new files with a specific stripe configuration.

Note – The file cannot exist prior to using `setstripe`. A directory must exist prior to using `setstripe`.

getstripe

Lists the striping information for a given filename or files in a directory or recursively for all files in a directory tree. It can also be used to list the files that have objects on a specific OST. Use one of these options:

```
[--quiet|-q] [--verbose|-v] [--recursive|-r]
```

find:

Search the directory tree rooted at the given directory or filename for files that match the given parameters.

Option	Description
--atime	The file was last accessed N*24 hours ago.
--ctime	The status of the file was last changed N*24 hours ago.
--mtime	The data in the file was last modified N*24 hours ago.
--obd	File has an object on a specific OST.
--maxdepth	Allows the <code>find</code> command to descend at most N levels of the directory tree.
--name	Returns only filenames that match "pattern", which is a shell pattern. Use "*" to match characters (except the path separator /), and '.' matches only '.', Use "!" --name" to exclude filenames matching the pattern.
--print	Prints the full filename, followed by a new line.
--print0	Prints the full filename, followed by a null (\0) character. Programs like <code>xargs -0</code> can be used to process filenames with spaces in them.
--ost	Returns only those files that have a stripe on the specified OST uuid. Currently, this can only specify a single OST UUID.

28.1.3 Examples

To create a file striped on two OSTs with 128 KB on each stripe, run:

```
$ lfs setstripe /mnt/lustre/file1 131072 -1 2
```

To delete default stripe pattern on directory, run:

```
$ lfs setstripe -d /mnt/lustre/dir
```

Note – New files use the default striping pattern created therein.

To list the object allocation of a given file, run:

```
$ lfs find /mnt/lustre/file1
```

To list the object allocations of all files in a given directory, run:

```
$ lfs find /mnt/lustre/
```

To list the objects of all files in a given directory tree recursively, run:

```
$ lfs find -r /mnt/lustre/
```

To list all files in a given directory that have objects on OST2-UUID recursively, run:

```
$ lfs find -r --obd OST2-UUID /mnt/lustre/
```

To change file owner and group, run:

```
$ lfs quotachown -i /mnt/lustre
```

To check quota for user and group, run:

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

To turn on the quotas of user and group, run:

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

Note – This turns on quotas after making the check.

To turn off the quotas of user and group, run:

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

To set quotas of user `bob`: 1GB block quota and 10,000 file quota, run:

```
$ lfs setquota -u bob 0 1000000 0 10000 /mnt/lustre
```

To set grace times for user quotas: 1000 seconds for block quotas, 1 week and 4 days for inode quotas, run:

```
$ lfs setquota -t -u 1000 1w4d /mnt/lustre
```

To list quotas of user `bob', run:

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

To show grace times for user quotas on /mnt/lustre, run:

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

To check the status of all servers (MDT, OST), run:

```
$ lfs check servers
```

To list all the OSTs, run;

```
$ lfs osts
```

To list inode consumption per OST and MDT, run:

```
$ lfs df -i
```

stripe-size	If you pass a stripe-size of 0, the filesystem default stripe size is used. Otherwise, the stripe-size must be a multiple of 16 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 0).
stripe-count	If you pass a stripe-count of 0, the filesystem default number of OSTs is used. A stripe-count of -1 means that all available OSTs should be used.
Note on defaults	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).

This is an example of setting and getting stripes, run:

```
$ lfs > setstripe lustre.iso 0 -1 0
$ lfs > getstripe lustre.iso
OBDS:
0: ost1_UUID ACTIVE
1: ost2_UUID_2 ACTIVE
./lustre
obdidx  objid  objid  group
1       4      0x4    0
```

To list the object allocations of all the files in a given directory, run:

```
$ lfs find/mnt/lustre/
```

To list the object allocation of a given file, run:

```
$ lfs find/mnt/lustre/file1
```

To list the extended attributes of a given file, run:

```
$ lfs find /mnt/lustre/fool
      OBDS:
      O: OST_localhost_UUID
/mnt/lustre/fool
obdidx  objid      objid      group
0        1         0x1        0
```

To list the extended attributes of all files in a given directory, run:

```
$ lfs find /mnt/lustre/
fs find -r /mnt/lustre/
```

To recursively list objects of all the files in a given directory tree, run:

```
$ lfs find -r /mnt/lustre/
```

To recursively list all the files in a given directory that have objects on OST2-UUID, run:

```
$ lfs find -r --obd OST2-UUID /mnt/lustre/
```

To change the file owner and group, run:

```
$ lfs quotachown -i /mnt/lustre
```

To check the quota for a user and a group, run:

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

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 ignore the error if the file does not exist, run the following command. For example:

```
$ lfs quotachown -i {file|directory} /mnt/lustre
```

To check the disk space in available inodes consumed by individual MDS and OST, run:

```
$ lfs df -i /mnt/lustre
```

uuid	inodes	used	free	use%	mounted on
mds-1_uuid	53265600	28266	53237334	0	/mnt/lustre[MDT:0]
ost-1_uuid	24405606	41349	244054715	0	/mnt//lustre[OST:0]
ost-2_uuid	244056064	884	244055180	0	/mnt/lustre[OST:1]

To check the disk space in size available on individual MDS and OST, run:

```
$ lfs df -h /mnt/lustre
```

uuid	1k-blocks	used	free	use%	mounted on
mds-1_uuid	203.5M	12.1M	191.5M	5	/mnt/lustre[MDT:0]
ost-1_uuid	1.8G	384.7M	1.4G	20	/mnt//lustre[OST:0]
ost-2_uuid	1.8G	343.0M	1.5G	18	/mnt/lustre[OST:1]
ost-3_uuid	1.8G	332.2M	1.5G	18	/mnt/lustre[OST:2]

To list the quotas of a user, run:

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

To check the status of all the servers – MDS and OSTs, run:

```
$ lfs check servers
```

OSC_localhost.localdomain_OST_localhost_mds1 active.
OSC_localhost.localdomain_OST_localhost_MNT_localhost active.
MDC_localhost.localdomain_mds1_MNT_localhost active.

To check the status of all the servers – MDSs, run:

```
$ lfs check mds
```

To check the status of all the servers – OSTs, run:

```
$ lfs check ost
```

To list all of the OSTs, run:

```
$ lfs osts
```

OBDS:
O: OST_localhost_UUID

To list the logs of particular types, run:

```
$ lfs catinfo {keyword} [node name]
```

Keywords are: **config**, **deletions**

The node name must be provided when using the keyword **config**.

For example, run:

```
$ lfs catinfo {config|dele*tions}{mdsnode|ostnode}
```

To join the files, run:

```
$ lfs join <filename_A> <filename_B>
```

This feature allows two files to be concatenated without performing I/O. For this utility, the first file must have a file size in multiples of 4K bytes. When the user performs an operation, the first file becomes concatenated, while the second file remain unlinked.

28.2 lfsck

The e2fsprogs package contains an lfsck tool which does distributed coherency checking for the Lustre filesystem, after e2fsck has been run. In most cases, e2fsck is sufficient to repair any filesystem issues and lfsck is not required (at the small chance of having some leaked space in the filesystem). To avoid lengthy downtime, you can also run lfsck once Lustre is already started (with care).

28.2.1 Synopsis

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

Note – For the above example *<filesystem>* refers to the mount point of the Lustre filesystem (default is */mnt/lustre*).

The parameters and their meanings:

Parameter	Description
-n	Performs a read-only check; does not repair the filesystem.
-l	Puts orphaned objects into a lost+found directory in the root of the filesystem.
-d	Deletes orphaned objects from the filesystem. 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 mds_database_file	MDS database file created by running <code>e2fsck --mdsdb mds_database_file device</code> on the MDS backing device.
--ostdb ost1_database_file[,ost2_database_file,...]	OST database files created by running <code>e2fsck --ostdb ost_database_file device</code> on each OST backing device.

28.2.2 Description

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

1. Run 'e2fsck -f' on the individual MDS / OST that had problems to fix any local filesystem 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 filesystem (in case this is needed later). After this is complete, you can bring the filesystem 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 filesystem, otherwise you might corrupt your filesystem. The `mdsdb` file can grow fairly large, depending on the number of files in the filesystem (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 filesystem 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 lfscck
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 filesystem 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 filesystem 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 filesystem is busy, `lfsck` may report inconsistencies where none exist because of files and objects being created/removed after the database files were collected. Examine 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 filesystem, 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 – It is not possible to write to the holes of such files without having `lfsck` recreate the objects, so it is generally 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; but `lfsck` cannot, by itself, tell which one is correct.

28.3 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:

Server options	Description
extents	Use extent mapped files
mballoc	Use Lustre filesystem allocator (required)

Lustre 1.6 server options	Description
abort_recov	Abort recovery when starting a target (currently an <code>lconf</code> 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

28.4 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
```


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.

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/supplementary 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 "NONE", which means the current behavior is used. This can be set to `/usr/sbin/l_getgroups` to use the Lustre group-supplied upcall, which 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 the `l_getgroups` to look at an external database for supplementary groups information.
- To set the upcall, echo `{path} > /proc/fs/lustre/mds/{mdsname}/group_upcall` or add the `--group_upcall {path}` line to the `--add mds` entry in the `lmc` script. You can also override this at startup time with a `--group_upcall {path}` option to `lconf`.
- 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 will wait, 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];
};
```


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`](#)

30.1.1 `llapi_file_create`

Use `llapi_file_create` to set Lustre properties for a new file.

30.1.1.1 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);
```

30.1.1.2 Description

The `llapi_file_create()` function sets a file descriptor's Lustre striping information. The file descriptor is then accessed with `open()`.

Parameter	Description
<code>llapi_file_create()</code>	If the file already exists, this parameter returns to 'EEXIST'. If the stripe parameters are invalid, this parameter returns to 'EINVAL'.
<code>stripe_size</code>	This value must be an even multiple of system page size, as shown by <code>getpagesize()</code> . The default Lustre stripe size is 4MB.
<code>stripe_offset</code>	Indicates the starting OST for this file.
<code>stripe_count</code>	Indicates the number of OSTs that this file will be striped across.
<code>stripe_pattern</code>	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`

30.1.1.3 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.

30.1.2.1 Synopsis

```
int llapi_file_get_stripe(const char *path, struct lov_user_md *lum)
```

30.1.2.2 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.

Parameter	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.

30.1.3.1 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)
```

30.1.3.2 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.

Parameter	Description
<code>name</code>	The name of the file.
<code>flags</code>	This opens flags.
<code>mode</code>	This opens modes.
<code>stripe_size</code>	The stripe size of the file.
<code>stripe_offset</code>	The stripe offset (<code>stripe_index</code>) of the file.
<code>stripe_count</code>	The stripe count of the file.
<code>stripe_pattern</code>	The stripe pattern of the file

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 lctl ping – 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 QSNNet

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	<p>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:</p> <ul style="list-style-type: none">• 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.
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 (1,Wc)	Boolean that determines whether to enable IRQ affinity. 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).
nmbk_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.
rnr_cnt (6,Wc)	Low-level QP parameter. Only change it from the default value if so advised.

<code>rrr_nak_timer</code> (0x10,Wc)	Low-level QP parameter. Only change it from the default value if so advised.
<code>fmr_remaps</code> (1000)	Controls how often FMR mappings may be reused before they must be unmapped. Only change it from the default value if so advised
<code>cksum</code> (0,W)	Boolean that determines if messages (NB not RDMA's) 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
<code>n_connd</code> (4)	Sets the number of connection daemons. The default value is 4.
<code>min_reconnect_interval</code> (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'.
<code>max_reconnect_interval</code> (60,W)	Maximum connection retry interval (in seconds).
<code>timeout</code> (50,W)	Time (in seconds) that communications may be stalled before the LND completes them with failure.
<code>ntx</code> (64)	Number of "normal" message descriptors for locally-initiated communications that may block for memory (callers block when this pool is exhausted).
<code>ntx_nblk</code> (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.
<code>concurrent_peers</code> (1024)	Maximum number of queue pairs and, therefore, the maximum number of peers that the instance of the LND may communicate with.
<code>cksum</code> (0,W)	Boolean that determines whether messages (NB not RDMA's) 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 `rxn_npares` 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 `rxn_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 `rxn_npares` 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 ptlnd 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 (ptlnd.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 ptlnd traffic.
rxn_pages (64 * #cpus)	Number of pages in an RX buffer.
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 (ptlnd) 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 PTLND 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 ptlnd 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 ptlnd 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_PEERCREDS (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.

System Configuration Utilities (man8)

This chapter includes system configuration utilities and includes the following sections:

- [mkfs.lustre](#)
- [tunefs.lustre](#)
- [lctl](#)
- [mount.lustre](#)
- [New Utilities in Lustre 1.6](#)

32.1 mkfs.lustre

`mkfs.lustre` is a utility to format a disk for a Lustre service.

32.1.1 Synopsis

```
mkfs.lustre <target_type> [options] device
```

where *<target_type>* is one of the following:

- OST object storage target
- MDT metadata storage target
- MGS configuration management service - one per site. This service can be combined with one `--mdt` service by specifying both types.

32.1.2 Description

`mkfs.lustre` is used to format a disk device in order to use it as part of a Lustre filesystem. After formatting, a disk can be mounted to start the Lustre service defined by this command.

Option	Description
<code>--backfstype=fstype</code>	Force a particular format for the backing filesystem (like ext3, ldiskfs)
<code>--comment=comment</code>	Set user comment about this disk, ignored by Lustre
<code>--device-size=KB</code>	Set device size for loop devices.
<code>--failnode=nid,...</code>	Set the NIDs of a failover partner. This option can be repeated as desired.
<code>--fsname=filesystem_name</code>	The Lustre filesystem of which this service/node will be a part. Default filesystem name is lustre.
<code>--index=index</code>	Force a particular OST or MDT index.
<code>--mkfsoptions=opts</code>	Format options for the backing filesystem. For example, ext3 options could be set here.
<code>--mountoptions=opts</code>	Set permanent mount options, equivalent to the setting in <code>/etc/fstab</code>
<code>--mgsnode=nid,...</code>	Set the NIDs of the MGS node, required for all targets other than the MGS.
<code>--noformat</code>	Only print would be done; this does not affect the disk.
<code>--param key=value</code>	Set permanent parameter key to value. This option can be repeated as desired. Typical options might include:
<code>--param sys.timeout=40</code>	System obd timeout
<code>--param lov.stripe.size=2097152</code>	Default stripe size
<code>--param lov.stripe.count=2</code>	Default stripe count
<code>--param failover mode=failout</code>	Return errors instead of waiting for recovery.
<code>--quiet</code>	Print less information.
<code>--reformat</code>	Reformat an existing Lustre disk.
<code>--stripe-count-hint=stripes</code>	Used for optimizing MDT inode size.
<code>--verbose</code>	Print more information.

32.1.3 Examples

To create a filesystem with MGS and MDT combined on the same node (cfs21), run:

```
$ mkfs.lustre --fsname=testfs --mdt --mgs /dev/sda1
```

To create OST for filesystem testfs on any number of nodes using the above MGS, run:

```
$ mkfs.lustre --fsname=testfs --ost --mgsnode=cfs21@tcp0 /dev/sdb
```

To create standalone MGS on, say, node cfs22, run:

```
$ mkfs.lustre --mgs /dev/sda1
```

To create MDT for filesystem myfs1 on any node, using the above MGS, run:

```
$ mkfs.lustre --fsname=myfs1 --mdt --mgsnode=cfs22@tcp0 /dev/sda2
```

32.2 tuneefs.lustre

`tuneefs.lustre` is the utility to modify the information of Lustre configuration on a disk.

32.2.1 Synopsis

```
tuneefs.lustre [options] device
```

32.2.2 Description

`tunefs.lustre` is used to modify the 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 filesystem.



Caution – Changes made here will affect a filesystem only when the target is mounted the next time.

Options	Description
<code>--comment=comment</code>	Sets a user comment about this disk, ignored by Lustre.
<code>--erase-params</code>	Removes all previous parameter information.
<code>--failnode=nid,...</code>	Sets the NID(s) of a failover partner. This option can be repeated as desired.
<code>--fsname=filesystem_name</code>	The Lustre filesystem of which this service will be a part. Default is 'lustre'.
<code>--index=index</code>	Forces a particular OST or MDT index.
<code>--mountfsoptions=opts</code>	Sets permanent mount options, equivalent to setting in <code>/etc/fstab</code> .
<code>--mgs</code>	Adds a configuration management service to this target.
<code>--msgnode=nid,...</code>	Sets the NID(s) of the MGS node, required for all targets other than the MGS.
<code>--noformat</code>	Only prints what would be done; does not affect the disk.
<code>--nomgs</code>	Removes a configuration management service to this target.
<code>--quiet</code>	Prints less information.
<code>--verbose</code>	Prints more information.
<code>--writeconf</code>	Erases all configuration logs for the filesystem of which this target is a part.
	Caution - This may prove VERY dangerous. All clients and servers should be stopped.

32.2.3 Examples

To create a filesystem with MGS and MDT combined on the same node (cfs21) -

```
$ tuneefs.lustre --fsname=testfs --mdt --mgs /dev/sda1
```

To create OST for filesystem testfs on any number of nodes using the above MGS -

```
$ tuneefs.lustre --fsname=testfs --ost --mgsnode=cfs21@tcp0 /dev/sdb
```

To create standalone MGS on, say, node cfs22 -

```
$ tuneefs.lustre --mgs /dev/sda1
```

To create MDT for filesystem myfs1 on any node, using the above MGS -

```
$ tuneefs.lustre --fsname=myfs1 --mdt --mgsnode=cfs22@tcp0 /dev/sda2
```

32.3 lctl

`lctl` is a Lustre utility used for low level configurations of Lustre filesystem. It also provides low-level testing and manages Lustre network (LNET) information.

32.3.1 Synopsis

```
lctl
lctl --device <devno> <command [args]>
lctl --threads <numthreads> <verbose> <devno> <command [args]>
```

32.3.2 Description

`lctl` can be invoked in interactive mode by issuing the following commands.

```
$ lctl
lctl> help
```

The most common commands in `lctl` are in matching pairs - like `device` and `attach`, `detach` and `setup`, `cleanup` and `connect`, `disconnect` and `help` and `quit`. For a complete list of available commands, type `help` at the `lctl` prompt. For help on command meaning and syntax, type `help command`. Command completion is activated by the TAB key. Command history is available via the “UP” and “DOWN” arrow keys.

For non-interactive single threaded use, use the second invocation, which runs commands after connecting to the device. Some commands are only used when specific functionality inside Lustre is tested, and not normally invoked by users. These commands are identified by the string (*CFS Dev*). Several commands are old and will be removed in the next major Lustre release. These commands are identified with the string (*Old*).

Network-Related Options	Description
--net <tcp/elan/myrinet>	The network type is to be used for the operation.
network <tcp/elan/myrinet>	Indicates what kind of network is applicable for the configuration commands that follow.
interface_list	Displays the interface entries and requires the 'network' command.
list_nids	Displays network identifiers (NIDs) defined on this node.
which_nid <remote host>	Identifies the path to a specific host by NID. Can be used to verify network setup and connectivity.
add_interface	Adds an interface entry. (<i>Old</i>)
del_interface [ip]	Deletes an interface entry. (<i>Old</i>)
peer_list	Displays the peer entries.
add_peer <nid> <host> <port>	Adds a peer entry. (<i>Lustre Development</i>)
del_peer <nid> <host> <port>	Removes a peer entry. (<i>Lustre Development</i>)
conn_list	Displays all the connected remote NIDs.
disconnect <nid>	Disconnects from a remote NID. (<i>Lustre Development</i>)
active_tx	Displays active transmits; used only for the Elan network type.
mynid [nid]	Informs the socknal of the local NID. It defaults to hostname for TCP networks, and is automatically set up for Elan/Myrinet networks. (<i>Lustre Development</i>)
add_uuid <uuid> <nid>	Associates a given UUID with an NID. (<i>Lustre Development</i>)
close_uuid <uuid>	Disconnects a UUID.
del_uuid <uuid>	Deletes a UUID association. (<i>Lustre Development</i>)
add_route <gateway> <target> [target]	Adds an entry to the routing table for the given target. (<i>Old</i>)
del_route <target>	Deletes an entry for a target from the routing table. (<i>Old</i>)

set_route <gateway> <up/down> [<time>]	Enables/disables routes via the given gateway in the portals routing table. <time> is used to specify when a gateway should come back online (Old)
route_list	Displays the complete routing table.
fail nid _all_[count]	Fails/restores communications. Omitting the count implies an indefinite fail. A count of zero indicates that communication should be restored. A non-zero count indicates the number of LNET messages to be dropped after which the communication is restored. The argument "nid" is used to specify the gateway, which is one peer of the communication. (<i>Lustre Development</i>)
show_route	Displays the complete routing table, same output as route_list.
ping nid [timeout] [pid]	Checks LNET connectivity, outputs a NIDs list on the target machine.

Device Selection	Description
newdev	Creates a new device.
device	Selects the specified OBD device. All other commands depend on the device being set.
cfg_device	Sets the current device being configured to <\$name>. (Old)
device_list	Shows all devices.
lustre_build_version	Displays the Lustre build system.

Device Configuration	Description
attach type [name [uuid]]	Attaches a type to the current device (which is set using the device command), and gives that device a name and a UUID. This allows us to identify the device for later use, and to know the type of that device.
setup cfg_device name	Types specific device setup commands. For obdfilter, a setup command tells the driver which block device it should use for storage and what type of filesystem is on that device.
cleanup cfg_device name	Cleans up a previously-setup device.
“detach” cfg_device	Removes a driver (and its name and UUID) from the current drive.
lov_setup lov-uuid stripe-count stripe-size offset pattern	

lov_getconfig <mountpoint> Reads LOV configuration from an MDS device. Returns default-stripe-count, default-stripe-size, offset, pattern, and a list of OST UUIDs. *(Old)*

lov_modify_tgts add|del
<lov-name> <uuid> <index>
<gen>

record cfs-uuid-name Records the commands that follow in the log.
endrecord Stops recording.
parse config-uuid-name Parse the log of recorded commands for a configuration.
dump_log config-uuid-name Displays the log of recorded commands for a config to kernel debug log.
clear_log config-name Deletes the current configuration log of recorded commands.

Device Operations	Descriptions
probe [timeout]	Builds a connection handle to a device. This command is used to suspend configuration until the lctl command ensures the availability of the MDS and OSC services. This avoids mount failures in a rebooting cluster.
close	Closes the connection handle.
getattr <objid>	Gets the attributes for an OST object <objid>. (<i>Lustre Development</i>)
setattr <objid> <mode>	Sets the mode attribute for an OST object <objid>. (<i>Lustre Development</i>)
create [num [mode [verbose [lsm data]]]]	Creates the specified number <num> of OST objects with the given <mode> (<i>Lustre Development</i>)
test_setattr <num> [verbose [[t]objid]]	
add_conn <conn_uuid> [priority]	
del_conn <conn_uuid>	
destroy <num>	Starting at <objid>, destroys <num> number of objects starting from the object with object id <objid> (<i>Lustre Development</i>)
test_getattr <num> [verbose [[t]objid]]	Does <num> getattrs on an OST object <objid> (objectid+1 on each thread) (<i>Lustre Development</i>)
test_brw [t]<num> [write [verbose [npages [[t]objid]]]]	Does <num> bulk read/ writes on an OST object <objid> (<npages> per I/O) (<i>Lustre Development</i>)
dump_ldlm	Dumps all the lock manager states. This is very useful for debugging
activate	Activates an import.

deactivate	Deactivates an import.
recover	<connection UUID>
lookup <directory> <file>	Displays the information of the given file.
notransno	Disables the sending of committed transnumber updates.
readonly	Disables writes to the underlying device.
abort_recovery	Aborts recovery on the MDS device.
mount_options	Dumps mount options to a file.
get_stripe	Shows stripe information for an echo client object.
set_stripe <objid>[width!count[@offset] [:id:id....]	Sets stripe information for an echo client object.
unset_stripe <objid>	Unsets stripe information for an echo client object.
del_mount_option profile	Deletes a specified profile.
set_timeout <secs>	Sets the timeout (obd_timeout) for a server to wait before failing recovery.
set_lustre_upcall </full/path/to/upcall>	Sets the Lustre upcall (obd_lustre_upcall) via the lustre.upcall sysctl.
llog_catlist	Lists all catalog logs on the current device.
llog_info <\$logname #oid#ogr#ogen>	Displays the log header information.
llog_print <\$logname #oid#ogr#ogen> [from] [to]	Displays the log content information. By default, it displays all records from index 1.
llog_check <\$logname #oid#ogr#ogen> [from] [to]	Checks the log content information. By default, it checks all records from index 1.
llog_cancel catalog id catalog name> <log id> <index>	Cancels a record in the log.
llog_remove <catalog id catalog name> <log id>	Removes a log from the catalog and erases it from the disk.

Debug	Description
debug_daemon	Debugs the daemon control and dumps to a file.
debug_kernel [file] [raw]	Gets the debug buffer and dumps to a file.
debug_file <input> <output>	Converts the kernel-dumped debug log from binary text format.

<code>clear</code>	Clears the kernel debug buffer.
<code>mark <text></code>	Inserts marker text in the kernel debug buffer.
<code>filter <subsystem id/debug mask></code>	Filters message type from the kernel debug buffer.
<code>show <subsystem id/debug mask></code>	Shows the specific type of message.
<code>debug_list <subs/types></code>	Lists all subsystem and debug types.
<code>modules <path></code>	Provides gdb-friendly module information.
<code>panic</code>	Forces the kernel to panic
<code>lwt start/stop [file]</code>	Lightweight tracing.
<code>memhog <page count> [<gfp flags>]</code>	Memory-pressure testing.

Control	Description
<code>help</code>	Shows a complete list of commands. <code>help <command name></code> can be used to get help on a specific command
<code>exit</code>	Closes the lctl session.
<code>quit</code>	Closes the lctl session.

Options (that can be used to invoke lctl)	Description
<code>--device</code>	The device number to be used for the operation. The value of devno is an integer, normally found by calling <code>lctl name2dev</code> on a device name.
<code>--threads</code>	The numthreads variable is a strictly positive integer indicating the number of threads to be started. The devno option is used as above.
<code>--ignore_errors ignore_errors</code>	Ignores errors during the script processing.
<code>dump</code>	Saves ioctls to a file.

32.3.3 Examples

attach

```
$ lctl
lctl > newdev
lctl > attach obdfilter OBDDEV OBDUUID
lctl > dl
4 AT obdfilter OBDDEV OBDUUID 1
```

getattr

```
$ lctl
lctl > newdev
lctl > attach obdfilter OBDDEV OBDUUID
lctl > dl
4 AT obdfilter OBDDEV OBDUUID 1lctl > getattr 12
id: 12
grp: 0
atime: 1002663714
mtime: 1002663535
ctime: 1002663535
size: 10
blocks: 8
blksize: 4096
mode: 100644
uid: 0
gid: 0
flags: 0
obdflags: 0
nlink: 1
valid: ffffffff
inline:
obdmd:
lctl > disconnect
Finished (success)
setup
lctl > setup /dev/loop0 extN
lctl > quit
```

32.3.4 Network Commands

The example below shows how to use `lctl` for identifying interface information and peers that are up. In this case, we have one MDS (`ft2`) and two OSS nodes (`d1_q_0`, `d2_q_0`). First we display the interface information on the MDS, and then list MDS peers:

```
$ lctl > network tcp up
$ lctl > interface_list
ft2: (10.67.73.181/255.255.255.0) npeer 0 nroute 2

$ lctl > peer_list
12345-10.67.73.150@tcp [1] ft2->d2_q_0:988 #6
12345-10.67.73.160@tcp [1] ft2->d1_q_0:988 #6
```

To identify routes and check connectivity to another node:

```
# lctl list_nids
10.67.73.181@tcp
# lctl which_nid d1_q_0
10.67.73.160@tcp
lctl ping d1_q_0
12345-0@lo
12345-10.67.73.160@tcp
```

Note – `lctl ping <target>` prints all NIDs at the target, not the NIDs of nodes traversed on the way to the target.

`Which_nid` does a lookup of the NID, and attempts to expand it. `which_nid` does not care about the node state. In the example below, the machine 'dellap' is real, the machine 'bogus' and the IP '10.67.73.212' are fake.

```
# lctl which_nid bogus@tcp
Can't parse NID bogus@tcp
# lctl which_nid dellap@tcp
10.67.73.89@tcp
# lctl which_nid 10.67.73.212@tcp
10.67.73.212@tcp
# lctl which_nid 10.67.758.54@tcp
Can't parse NID 10.67.758.54@tcp
```

32.4 mount.lustre

`mount.lustre` is a utility that starts a Lustre client or target service.

32.4.1 Synopsis

```
$ mount -t lustre [-o options] device dir
```

32.4.2 Description

`mount.lustre` is used to start a Lustre client or target service. This program should not be called directly; rather it is a helper program invoked through `mount(8)`. Lustre clients and targets are stopped by using the `umount(8)` command.

There are two forms for the device option, depending on whether a client or a target service is started:

`<mgsspec>:/<fsname>`

This is a client mount command to mount the Lustre filesystem 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	Description
<code><mgsspec>:=<mgssnode>[:<mgssnode>]</code>	The mgs specification may be a colon-separated list of nodes...
<code><mgssnode>:=<mgssnid>[,<mgssnid>]</code>	...and 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:

Options	Descriptions
flock	Enable flock support.
noflock	Disable flock support.
user_xattr	Enable get/set user xattr.
nouser_xattr	Disable user xattr.
acl	Enable ACL support.
noacl	Disable ACL support.

In addition to the standard mount options and backing disk type (e.g. LDISKFS) options, Lustre understands the following server-specific options:

Options	Descriptions
nosvc	Only start the MGC (and MGS, if co-located) for a target service, and not the actual service.
exclude=ostlist	Start a client or MDT with a (colon-separated) list of known inactive OSTs
abort_recov	Abort recovery (targets only)

32.4.3 Examples

Mounting a client – no failover:

MDS nid is 10.10.0.5@tcp0

MDT is mds-p (specified by -mds in .xml file)

Mount point is /mnt/lustre

client is defined in the .xml file

```
# mount -t lustre 10.10.0.5@tcp0:/mds-p/client /mnt/lustre
```

Add a failover MDS at 10.10.0.6@tcp0:

```
# mount -t lustre 10.10.0.5@tcp0:10.10.0.6@tcp0:/mds-p/client \
/mnt/lustre
```

32.5 New Utilities in Lustre 1.6

This section describes new utilities available in Lustre 1.6.

32.5.1 General Purpose Utilities

The following utility is located in `/usr/bin`.

lustre_rmmod.sh

The `lustre_rmmod.sh` utility removes all Lustre and LNET modules, assuming no Lustre services are running.

Note – The `lustre_rmmod.sh` utility does not work if Lustre modules are being used or if you have manually fired the `lctl network up` command.

32.5.2 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](#).

32.5.3 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`.

32.5.4 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
```

32.5.5 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.

32.5.6 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 only consistent on local nodes. To enable it, use <code>-o localflock</code> . This is a client mount option.
consistent mode	In this mode, locks are consistent over all nodes in the filesystem. To enable it, use the <code>-o flock</code> . This is a client mount option.

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.

32.5.6.1 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
```


System Limits

This chapter describes various limits on the size of files and filesystems. 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:

- [Maximum Stripe Count](#)
- [Maximum Stripe Size](#)
- [Minimum Stripe Size](#)
- [Maximum Number of OSTs and MDSs](#)
- [Maximum Number of Clients](#)
- [Maximum Size of a Filesystem](#)
- [Maximum File Size](#)
- [Maximum Number of Files or Subdirectories in a Single Directory](#)
- [MDS Space Consumption](#)
- [Maximum Length of a Filename and Pathname](#)
- [Maximum Number of Open Files for Lustre Filesystems](#)
- [OSS RAM Size for a Single OST](#)

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 filesystem. 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 64KB `PAGE_SIZE` on some 64-bit machines, the minimum stripe size is set to 64 KB.

33.4 Maximum Number of OSTs and MDSs

You can set the maximum number of OSTs by a compile option. The limit of 512 OSTs in Lustre 1.4.6 is raised to 1020 OSTs in Lustre releases 1.4.7 and later. Rigorous 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 32768. We have tested up to 22000 clients.

33.6 Maximum Size of a Filesystem

For i386 systems in 2.6 kernels, the block devices are limited to 16 TB. Each OST or MDS can have a filesystem up to 8 TB (For 2.6 kernels, the 8 TB limit is imposed by ext3). You can have multiple OST filesystems on a single node. Currently, the largest production Lustre filesystem has 448 OSTs in a single filesystem. There is a compile-time limit of 8150 OSTs in a single filesystem, giving a theoretical filesystem limit of nearly 64 PB.

Several production Lustre filesystems have around 100 OSSs in a single filesystem. The largest filesystem 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. You can have unlimited subdirectories in all the 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 filesystem with MDS of 2 TB. This can be increased initially, at the time of MDS filesystem 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 1KB 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 Filesystems](#).

33.10 Maximum Length of a Filename and Pathname

This limit is 255 bytes for a single filename, the same as in an `ext3` filesystem. The Linux VFS imposes a full pathname length of 4096 bytes.

33.11 Maximum Number of Open Files for Lustre Filesystems

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.

You can decrease the OST count by half for 4TB LUNs in the same timeframe you get failover (which will double the OST count).

Feature List

Networks

TCP

Elan

QSW

userspace tcp

userspace portals

Utilities

lctl

lfs

lfs getstripe

lfs setstripe

lfs find (lfind)

lfs check (lfsck)

mount.lustre

mkfs.lustre

Special System Cell Behavior

disabling POSIX locking

group locks

Modules

- INET
- acceptor
- accept_port
- accept_backlog
- accept_timeout
- accept_proto_version
- config_on_load

networks

- routes
- ip2nets
- forwarding (obsolete)
- implicit_loopback
- small_router_buffers
- large_router_buffers
- tiny_router_buffer

SOCKLND Kernel TCP/IP LND

- timeout
- nconnds
- min_reconnectms
- max_reconnectms
- eager_ack
- typed_conns
- min_bulk
- tx_buffer_size, rx_buffer_size
- nagle
- keepalive_idle
- keepalive_intvl
- keepalive_count
- enable_irq_affinity
- zc_min_frag

QSW LND

- tx_maxconfig
- ntxmsgs
- nblk_txmsg
- nrxmsg_small
- ep_envelopes_small
- nrxmsg_large
- ep_envelopes_large
- optimized_puts
- optimized_gets

RapidArray LND

- n_connd
- min_reconnect_interval
- max_reconnect_interval
- timeout
- ntx
- ntx_nblk
- fma_cq_size
- max_immediate

VIB LND

service_number
arp_retries
min_reconnect_interval
max_reconnect_interval
timeout
ntx
ntx_nblk
concurrent_peers
hca_basename
ipif_basename
local_ack_timeout
retry_cnt
rnr_cnt
rnr_nak_timer
fmr_remaps
cksum

OpenIB LND

n_connd
min_reconnect_interval
max_reconnect_interval
timeout
ntx
ntx_nblk
concurrent_peers
cksum

Portals LND (Linux)

- ntx
- concurrent_peers
- peer_hash_table_size
- cksum
- timeout
- portal
- rxb_npages
- credits
- peercredits
- max_msg_size

Portals LND (Catamount)

- PT LLND_PORTAL
- PT LLND_PID
- PT LLND_PEERCREDITS
- PT LLND_MAX_MESSAGE_SIZE
- PT LLND_MAX_MSGS_PER_BUFFER
- PT LND_MSG_SPARE
- PT LLND_PEER_HASH_SIZE
- PT LLND_EQ_SIZE

Lustre APIs

- User/Group Cache Upcall
- Striping using ioctls
- Direct I/O

Task List

Key Concepts

- software
- clients
- OSTs
- MDT
- data in /proc

User Tasks

- free space
- start servers
- change ACL
- getstripe
- setstripe
- Direct I/O
- flock
- group locks

Administrator Tasks

- Build
- Install
- new
- Downgrade
- Configure
- change configure
- change server IP
- migrate OST
- add storage
- grow disk
- add oss
- Stop - start
- mount / unmount (-force)
- init.d/lustre scripts
- failover by hand
- get status
- /proc
- /var/log/messages
- Tuning

Architect Tasks

- Networking
- understand hardware options
- naming: nid's networks
- Multihomed servers
- routes

Version Log

Manual Version	Date	Details of Edits	Bug
1.11	3/11/08	1. Updated content in Failover chapter.	12143
		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 filesystems.	14696
		17. Missing build instructions.	14913
		18. Update ip2nets section in Lustre manual and add example shown	12382

Manual Version	Date	Details of Edits	Bug
1.10	12/18/07	19. Free space management	12175
		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
		6. Updated unlink and munlink command information in the Identifying a Missing OST topic in the Lustre Troubleshooting and Tips chapter.	14239
1.9	11/2/07	7. Minor error in manual Chapter III - 3.2.3.3	14414
		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

Manual Version	Date	Details of Edits	Bug
		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 chapter.	13810/ 11325/ 12164
		17. Updated content in Prerequisites and Lustre Installation chapters.	13851
		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 mballo3 content to the LustreProc chapter.	12384/ 10816
1.6	08/23/07	1. Updated content in the Expanding the Filesystem by Adding OSTs section.	13118

Manual Version	Date	Details of Edits	Bug
		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
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

Manual Version	Date	Details of Edits	Bug
		2. Added enhancements to the DDN Tuning chapter.	12173
		3. Updated the User Utilities (man1) chapter.	n/a
		4. Added Ifsck 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
		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 tuneufs.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.	

Lustre Knowledge Base

The Knowledge Base is a collection of tips and general information regarding Lustre.

[How to reclaim the 5 percent of disk space reserved for root?](#)

[Why are applications hanging?](#)

[How do I abort recovery? Why would I want to?](#)

[What does "denying connection for new client" mean?](#)

[How do I set a default debug level for clients?](#)

[How can I improve Lustre metadata performance when using large directories \(> 0.5 million files\)?](#)

[Filesystem refuses to mount because of UUID mismatch](#)

[How do I set up multiple Lustre filesystems on the same node?](#)

[Is it possible to change the IP address of a OST? MDS? Change the UUID?](#)

[How do I replace an OST or MDS?](#)

[How do I configure recoverable / failover object servers?](#)

[How do I resize an MDS / OST filesystem?](#)

[How do I backup / restore a Lustre filesystem?](#)

[How do I control multiple services on one node independently?](#)

[What extra resources are required for automated failover?](#)

[Is there a way to tell which OST a process on a client is using?](#)

[I need multiple SCSI LUNs per HBA - what is the best way to do this?](#)

[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 filesystem 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?

How to reclaim the 5 percent of disk space reserved for root?

If your filesystem 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 filesystem 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 3 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 portals.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).

Filesystem 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 ostonode --lov lov1 --dev /dev/sdc --ostuuid
3dbf8_OST_ostonode_ddd780786b
```

```
lmc -add mds --node mdsnode --mds mds_scratch --dev /dev/sdc --mdsuuid
8a9c5_mds_scratch_8d2422aa88
```

How do I set up multiple Lustre filesystems on the same node?

Assuming you want to have separate filesystems with different mount locations, you need a dedicated MDS partition and Logical Object Volume (LOV) for each filesystem. 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 filesystem "foo" already exists, and you want to add the filesystem "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 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 filesystem 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 filesystem is simply a normal ext3 filesystem, 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 filesystems. For example:

```
mount -t ext3 /dev/old /mnt/ost_old
mount -t ext3 /dev/new /mnt/ost_new
rsync -aV /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 -aVz /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 filesystem?

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 filesystem, 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 filesystems. If these filesystems 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 filesystem to use this increased space.

1. **Prior to doing any sort of low-level changes like this, back up the filesystem and/or device. See [How do I backup / restore a Lustre filesystem?](#)**
2. **After the filesystem 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 filesystem, using the Lustre e2fsprogs (available from the Lustre download site or <http://ftp.lustre.org/other/e2fsprogs>). Run:**

```
e2fsck -f {dev}
```

4. **Resize the filesystem to use the increased size of the device. Run:**

```
resize2fs -p {dev}
```

How do I backup / restore a Lustre filesystem?

Several types of Lustre backups are available.

CLIENT FILE SYSTEM-LEVEL BACKUPS

It is possible to back up Lustre filesystems 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 filesystems, full backups are not always possible. Doing backups of subsets of the filesystem (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 filesystem is very robust and it may be possible to recover filesystem 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 filesystem 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 filesystem need to be changed, to use less space for the backup, etc).

In this case it is possible to mount the ext3 filesystem directly from the storage device and do a file-level backup. Lustre **MUST BE STOPPED ON THAT NODE**.

To back up such a filesystem properly also requires that any extended attributes (EAs) stored in the filesystem 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 filesystem.

2. Mount the filesystem there.

- For 2.4 kernels use: `mount -t ext3 {dev} /mnt/mds`
- For 2.6 kernels use: `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.lov0s0AvRCwEAAABXoKUCAAAAAAAAAAAAAAAAAAAAAQAEEAAADD5QoAAAAA
AAAAAAAAAAAAAAAAAAAAEAAAA=
```

6. Back up all filesystem data. Type:

```
tar czvf {backup file}.tgz
```

7. Change out of the mounted filesystem. Type.

```
cd -
```

8. Unmount the filesystem. Type:

```
umount /mnt/mds
```

Follow the same process on each of the OST device filesystems. 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.

1. **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 filesystems, 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 filesystems, use: `mke2fs -j -J size=400 -I 256 -i 16384 {dev}`

2. **Enable ext3 filesystem directory indexing. Type:**

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

3. **Mount the filesystem. Type:**

- For 2.4 kernels use: `mount -t ext3 {dev} /mnt/mds`
- For 2.6 kernels use: `mount -t ldiskfs {dev} /mnt/mds`

4. **Change to the new filesystem mount point. Type:**

```
cd /mnt/mds
```

5. **Restore the filesystem backup. Type:**

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

6. **Restore the filesystem EAs. Type:**

```
setfattr --restore=ea.bak
```

7. **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 filesystem was used between the time the backup was made and when it was restored, then the "fsck" tool (part of Lustre e2fsprogs) can be run to ensure the filesystem is coherent. If all of the device filesystems were backed up at the same time after the whole Lustre filesystem was stopped this is not necessary. The filesystem should be immediately usable even if fsck 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 a process on a client is using?

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, i386 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.

■ **Voltaire**

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 filesystem 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 filesystem 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 it is advisable to mark the missing OST unavailable so clients and the MDS do not time out trying to contact it. On MDS+client nodes:

```
# lctl dl      # to generate a list of devices, find the OST device number
# lctl --device N deactivate # N will be different between the MDS
and clients
```

If the OST later becomes available it needs to be reactivated:

```
# lctl --device N activate
```

Determine all the files striped over the missing OST:

```
# lfs find -R -o {OST_UUID} /mountpoint
```

This returns a simple list of filenames from the affected filesystem.

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 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:

$$[(C*N + X)*S, (C*N + X)*S + S - 1], N = \{ 0, 1, 2, \dots \}$$

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:

$$[(2*N + 0)*1M, (2*N + 0)*1M + 1M - 1], N = \{ 0, 1, 2, \dots \}$$

If the filesystem 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 filesystem 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 filesystem 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 filesystem 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 filesystem 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 -ls /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 filesystem pages in memory and memory pressure. A kernel thread is woken to flush dirty pages to the filesystem, 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.

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. 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 that 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 scheduling 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 metadata network protocol.
Metadata Write-back Cache	A cache of metadata updates (mkdir, create, setattr, other operations) which an application has performed, but have not yet been flushed to a storage device or server. 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

Numerics

1.6 utilities, 32-15

A

access control list (ACL), 26-1

ACL, using, 26-1

ACLs

 examples, 26-3

 Lustre support, 26-2

active / active configuration, failover, 8-6

adding multiple LUNs on a single HBA, 27-4

B

backing up

 MDS file, 15-3

 OST file, 15-4

backup

 device-level, 15-2

 file-level, 15-2

 filesystem-level, 15-1

backup and restore, 15-1

benchmark

 Bonnie++, 17-2

 IOR, 17-3

 IOzone, 17-5

bonding, 12-1

 configuring Lustre, 12-11

 module parameters, 12-5

 references, 12-11

 requirements, 12-2

 setting up, 12-5

bonding NICs, 12-4

Bonnie++ benchmark, 17-2

building, 14-2

building a kernel, 4-14

 installing Quilt, 4-15

 selecting a patch series, 4-14

 using Quilt, 4-15

building a tarball, 4-21

building from source, 4-20

building Lustre, 4-16

 configuration options, 4-19

 liblustre, 4-19

building the Lustre SNMP module, 14-2

C

client node, mounting Lustre, 4-7

client read/write

 extents survey, 20-11

 offset survey, 20-10

command

 lfsck, 28-8

 mount, 28-14

command lfs, 28-1

complicated configurations, multihomed servers, 7-1

components, Lustre, 1-4

configuration

 abort recovery, 4-13

 changing a server NID, 4-13

 failout, 4-10

 failover, 4-9

 filesystem name, 4-7

 making/starting a filesystem, 4-5

- module setup, 4-4
- mount with inactive OSTs, 4-10
- removing an OST, 4-12
- running multiple Lustres, 4-11
- stopping a server, 4-8
- without Lustre service, 4-10
- writeconf, 4-13
- configuration
 - starting a server, 4-8
- configuring a Lustre network, 5-4
- configuring Lustre, 4-3
- configuring Lustre network, 5-4
- consistent clocks, 3-7
- core-required tools, 3-5
- COW I/O, 19-12
- D**
- DDN tuning, 21-6
 - setting maxcmds, 21-8
 - setting readahead and MF, 21-7
 - setting segment size, 21-7
 - setting write-back cache, 21-8
- debugging
 - adding debugging to source code, 23-10
 - controlling the kernel debug log, 23-6
 - daemon, 23-5
 - debugging in UML, 23-10
 - finding Lustre UUID of an OST, 23-13
 - finding memory leaks, 23-8
 - lctl tool, 23-7
 - looking at disk content, 23-12
 - LWT tracing, 23-15
 - messages, 23-2
 - printing to /var/log/messages, 23-9
 - Ptlrpc request history, 23-14
 - sample lctl run, 23-9
 - tcpdump, 23-13
 - tools, 23-4
 - tracing lock traffic, 23-9
 - using strace for troubleshooting, 23-11
- debugging tools, 3-6
- designing a Lustre network, 5-1
- device-level backup, 15-2
- device-level restore, 15-4
- DIRECT I/O, 19-12
- downgrade
 - filesystem, 13-9

- Lustre version 1.6.3 to version 1.4.11, 13-8
- requirements, 13-8
- downloading Lustre, 3-2

E

- Elan (Quadrics Elan), 2-2
- Elan to TCP routing
 - modprobe.conf, 7-4
 - start clients, 7-5
 - start servers, 7-5
- end-to-end client checksums, 25-11
- environmental requirements
 - changing kernel I/O schedulers, 3-8
 - choosing kernel I/O schedulers, 3-7
 - consistent clocks, 3-7
 - SSH access, 3-6
 - universal UID/GID, 3-7
- error messages, 22-1
- Ethernet, 2-2

F

- failover, 8-1
 - active / active configuration, 8-6
 - configuring, 4-9
 - configuring MDS and OSTs, 8-6
 - connection handling, 8-4
 - hardware requirements, 8-7
 - Heartbeat, 8-3
 - MDS, 8-6
 - OST, 8-5
 - power equipment, 8-3
 - power management software, 8-3
 - role of nodes, 8-5
 - setup with Heartbeat V1, 8-8
 - setup with Heartbeat V2, 8-19
 - software, considerations, 8-24
 - starting / stopping a resource, 8-6
- failover, Heartbeat V1
 - configuring Heartbeat, 8-9
 - installing software, 8-8
 - Mon setup, 8-15
- failover, Heartbeat V2
 - configuring hardware, 8-20
 - installing software, 8-19
 - operating, 8-23
- file striping, 25-1
- file-level backup, 15-2

- filesystem
 - making/starting, 4-5
 - name, 4-7
- filesystem-level backup, 15-1
- flock utility, 32-19
- free space
 - querying, 24-1
- free space management
 - adjusting weighting between free space and location, 25-9
 - round-robin allocator, 25-8
 - weighted allocator, 25-9

G

- GID, 3-7
- gm (Myrinet), 2-2
- group ID (GID), 3-7

H

- HA software, 3-5
- handling timeouts, 28-15
- HBA, adding SCSI LUNs, 27-4
- Heartbeat configuration
 - with STONITH, 8-12
 - without STONITH, 8-9
- Heartbeat V1, failover setup, 8-8
- Heartbeat V2, failover setup, 8-19

I

- I/O options
 - end-to-end client checksums, 25-11
- I/O tunables, 20-7
- improving Lustre metadata performance with large directories, 27-5
- Infinicon InfiniBand (iib), 2-2
- installing, 14-2
 - POSIX, 16-2
- installing Lustre, 4-2
- installing Lustre, required software
 - core-required tools, 3-5
 - debugging tools, 3-6
 - HA software, 3-5
- installing Quilt, 4-15
- installing the Lustre SNMP module, 14-2
- interoperability, lustre, 13-1

- IOR benchmark, 17-3
- IOzone benchmark, 17-5

K

- Kerberos
 - Lustre setup, 11-2
 - Lustre-Kerberos flavors, 11-9
- kernel
 - building, 4-14
 - I/O schedulers, changing, 3-8
 - I/O schedulers, choosing, 3-7
- kernel-modules.rpm, 3-4
- kernel-smp.rpm, 3-4
- kernel-source.rpm, 3-4

L

- lctl, 32-5
 - lustre.rpm, 3-5
- lctl tool, 23-7
- lfs
 - lustre.rpm, 3-5
- lfs command, 28-1
- lfs getstripe
 - display files and directories, 25-4
 - setting striping patterns, 25-6
- lfsck command, 28-8
- liblustre, 4-19
- llog_reader utility, 32-18
- llstat.sh utility, 32-16
- LND, 2-1
- LNET, 2-2
 - starting, 5-10
 - stopping, 5-11
- loadgen utility, 32-18
- locking proc entries, 20-18
- lockless tunables, 21-12
- logs, 22-1
- lr_reader utility, 32-18
- LUNs, adding, 27-4
- Lustre
 - building, 4-16
 - building a tarball, 4-21
 - building from source, 4-20
 - configuring, 4-3
 - installing, 4-2

- memory requirements, 3-9
- recovering, 18-1
- lustre
 - downgrading, 13-1
 - interoperability, 13-1
 - upgrading, 13-1
- Lustre client node, 1-6
- Lustre I/O kit
 - downloading, 19-2
 - obdfilter_survey tool, 19-4
 - ost_survey tool, 19-9
 - PIOS I/O modes, 19-12
 - PIOS tool, 19-10
 - prerequisites to using, 19-2
 - running tests, 19-2
 - sgpdd_survey tool, 19-3
- Lustre Network Driver (LND), 2-1
- Lustre SNMP module, 14-2, 14-3
- Lustre software
 - Lustre modules, 1-2
 - patched Linux kernel, 1-2
 - userspace utilities, 1-3
- lustre-.rpm, 3-4
 - lctl, 3-5
 - lfs, 3-5
 - mkfs.lustre, 3-5
 - mount.lustre, 3-5
- lustre_config.sh utility, 32-15
- lustre_createcsv.sh utility, 32-15
- lustre_req_history.sh utility, 32-16
- lustre_rmmod.sh utility, 32-15
- lustre_up14.sh utility, 32-16
- lustre-ldiskfs-.rpm, 3-4
- lustre-source-.rpm, 3-4

M

- man1
 - lfs, 28-1
 - lfsck, 28-8
 - mount, 28-14
- man3
 - user/group cache upcall, 29-1
- man5
 - LNET options, 31-3
 - module options, 31-2
 - MX LND, 31-18

- OpenIB LND, 31-13
- Portals LND (Catamount), 31-16
- Portals LND (Linux), 31-14
- QSW LND, 31-10
- RapidArray LND, 31-11
- VIB LND, 31-12
- man8
 - extents_stats utility extents_stats utility, 32-17
 - lctl, 32-5
 - llog_reader utility, 32-18
 - llstat.sh, 32-16
 - loadgen utility, 32-18
 - lr_reader utility, 32-15
 - lustre_config.sh, 32-15
 - lustre_createcsv.sh utility, 32-15
 - lustre_req_history.sh, 32-16
 - lustre_rmmod.sh utility, 32-15
 - lustre_up14.sh utility, 32-16
 - mkfs.lustre, 32-1
 - mount.lustre, 32-13
 - network commands, 32-12
 - offset_stats utility, 32-17
 - plot-llstat.sh, 32-16
 - tunefs.lustre, 32-3
 - vfs_ops_stats utility vfs_ops_stats utility, 32-17
- Management Server (MGS), 1-5
- mballoc
 - history, 20-14
- mballoc3
 - tunables, 20-16
- MDS
 - failover, 8-6
 - failover configuration, 8-6
 - memory, determining, 3-9
- MDS file, backing up, 15-3
- MDS/OST formatting
 - calculating MDT size, 21-4
 - overriding default formatting options, 21-5
 - planning for inodes, 21-4
- MDT, 1-6
- Mellanox-Gold InfiniBand, 2-2
- memory requirements, 3-9
- Metadata Target (MDT), 1-6
- MGS, 1-5
- mkfs.lustre, 32-1
 - lustre-.rpm, 3-5
- MMP, using, 8-18

- mod5
 - SOCKLND kernel TCP/IP LND, 31-8
- modprobe.conf, 7-1, 7-4
- module setup, 4-4
- mount command, 28-14
- mount.lustre, 32-13
 - lustre-.rpm, 3-5
- MountConf, 4-2
- multihomed server
 - Lustre complicated configurations, 7-1
 - modprobe.conf, 7-1
 - start clients, 7-4
 - start server, 7-3
- multiple mount protection, see MMP, 8-18
- multiple NICs, 12-4
- MX LND, 31-18
- Myrinet, 2-2

N

- network
 - bonding, 12-1
- network identifier (NID), 2-2
- networks, supported
 - Elan (Quadrics Elan), 2-2
 - gm (Myrinet), 2-2
 - iib (Infinicon InfiniBand), 2-2
 - o2ib (OFED), 2-2
 - openlib (Mellanox-Gold InfiniBand), 2-2
 - ra (RapidArray), 2-2
 - tcp (Ethernet), 2-2
 - vib (Voltaire InfiniBand), 2-2
- NIC
 - bonding, 12-4
 - multiple, 12-4
- NID, 2-2
- NID, server, changing, 4-13
- node
 - active / active, 8-5
 - active / passive, 8-5

O

- o2ib (OFED), 2-2
- obdfilter_survey tool, 19-4
- Object Storage Target (OST), 1-6
- OFED, 2-2
- offset_stats utility, 32-17

- OpenIB LND, 31-13
- openlib (Mellanox-Gold InfiniBand), 2-2
- operating tips
 - adding OSTs, 27-1
 - data migration script, simple, 27-3
- OSS
 - memory, requirements, 3-10
- OST, 1-6
 - adding, 27-1
 - failover, 8-5
 - failover configuration, 8-6
- OST block I/O stream, watching, 20-12
- OST file, backing up, 15-4
- ost_survey tool, 19-9
- overview, Lustre, 1-1

P

- patch series, selecting, 4-14
- performance tips, 22-2
- performing direct I/O, 25-10
- PIOS
 - examples, 19-16
- PIOS I/O mode
 - COW I/O, 19-12
 - DIRECT I/O, 19-12
 - POSIX I/O, 19-12
- PIOS I/O modes, 19-12
- PIOS parameter
 - ChunkSize(c), 19-13
 - Offset(o), 19-14
 - RegionCount(n), 19-13
 - RegionSize(s), 19-13
 - ThreadCount(t), 19-13
- PIOS tool, 19-10
- plot-llstat.sh utility, 32-16
- Portals LND
 - Catamount, 31-16
 - Linux, 31-14
- POSIX
 - debugging, VSX_DEBUG_FILE=output_file, 16-5
 - debugging, VSX_DEBUG_FLAGS=xxxxxx, 16-5
 - installing, 16-2
 - running tests against Lustre, 16-4
- POSIX I/O, 19-12
- power equipment, 8-3
- power management software, 8-3

pre-packaged releases, Lustre, 3-4

proc entries

- debug support, 20-19

- introduction, 20-2

- locking, 20-18

Q

QSW LND, 31-10

Quadrics Elan, 2-2

querying filesystem space, 24-1

quotas

- administering, 9-3

- allocating, 9-5

- configuring, 9-2

- creating files, 9-3

- reset, 9-4

- working with, 9-1

R

ra (RapidArray), 2-2

RAID

- considerations for backend storage, 10-1

- creating an external journal, 10-12

- disk performance management, 10-4

- formatting, 10-4

- performance considerations, 10-3

- selecting storage for the MDS and OSS, 10-2

- understanding double failures with RAID5

 - hardware and software, 10-3

RapidArray, 2-2

RapidArray LND, 31-11

readahead, using, 20-13

recovering Lustre, 18-1

recovery mode, failure types

- client failure, 18-2

- MDS failure/failover, 18-2

- network partition, 18-3

- OST failure, 18-3

reset quota, 9-4

restore

- device-level, 15-4

round-robin allocator, 25-8

routing, elan to TCP, 7-4

RPC stream tunables, 20-7

RPC stream, watching, 20-9

RPM packages, 3-4

kernel-modules-.rpm, 3-4

kernel-smp-.rpm, 3-4

kernel-source-.rpm, 3-4

lustre-.rpm, 3-4

lustre-ldiskfs-.rpm, 3-4

lustre-source-.rpm, 3-4

running a client and OST on the same machine, 27-5

S

server

- starting, 4-8

- stopping, 4-8

server NID, changing, 4-13

setting

- maxcmds, 21-8

- readahead and MF, 21-7

- SCSI I/O sizes, 22-2

- segment size, 21-7

- write-back cache, 21-8

sgpdd_survey tool, 19-3

simple configuration

- CSV file, configuring Lustre, 6-4

- network, combined MGS/MDT, 6-1

- network, separate MGS/MDT, 6-3

- TCP network, Lustre simple configurations, 6-1

SOCKLND kernel TCP/IP LND, 31-8

SSH access, 3-6

starting

- LNET, 5-10

stopping

- LNET, 5-11

striping

- advantages, 25-2

- disadvantages, 25-3

- lfs getstripe, display files and directories, 25-4

- lfs getstripe, set striping pattern, 25-6

- size, 25-3

supported configurations, Lustre, 3-3

supported networks

- Elan (Quadrics Elan), 2-2

- gm (Myrinet), 2-2

- iib (Infinicon InfiniBand), 2-2

- o2ib (OFED), 2-2

- openlib (Mellanox-Gold InfiniBand), 2-2

- ra (RapidArray), 2-2

- tcp (Ethernet), 2-2

- vib (Voltaire InfiniBand), 2-2

T

tarball

- from source RPM, 4-21

tarball from CVS, 4-22

tcp (Ethernet), 2-2

timeouts, handling, 28-15

troubleshooting

- changing parameters, 22-6

- consideration in connecting a SAN with

 - Lustre, 22-8

- default striping, 22-7

- drawbacks in doing multi-client O_APPEND

 - writes, 22-12

- erasing a file system, 22-7

- error messages, 22-1

- handling timeouts on initial Lustre setup, 22-11

- handling/debugging "bind

 - address already in use" error, 22-8

- handling/debugging "Lustre Error

 - xxx went back in time", 22-11

- handling/debugging error "28", 22-10

- identifying a missing OST, 22-4

- log message 'out of memory' on OST, 22-13

- logs, 22-1

- Lustre Error

 - "slow start_page_write", 22-12

- OST object missing or damaged, 22-3

- OSTs become read-only, 22-4

- reclaiming reserved disk space, 22-8

- replacing an existing OST or MDS, 22-9

- setting SCSI I/O sizes, 22-2

- slowdown occurs during Lustre startup, 22-13

- triggering watchdog for PID NNN, 22-10

- write performance better than read

 - performance, 22-2

tunables

- RPC stream, 20-7

tunables, lockless, 21-12

tunefs.lustre, 32-3

tuning

- DDN, 21-6

- formatting the MDS and OST, 21-4

- large-scale, 21-10

- LNET tunables, 21-3

- module options, 21-1

- module threads, 21-2

U

UID, 3-7

universal UID/GID, 3-7

upgrade

- multiple filesystems (shared MGS), 13-7

- requirements, 13-2

- single filesystem, 13-4

- supported paths, 13-3

- version 1.4.11 (and later) to version 1.6.3, 13-2

upgrading

- starting clients, 13-4

user ID (UID), 3-7

using, 14-3

- quotas, 24-3

using Quilt, 4-15

using the Lustre SNMP module, 14-3

utilities

- new, v1.6, 32-15

V

VIB LND, 31-12

Voltaire InfiniBand (vib), 2-2

VSX_DEBUG_FILE=output_file, 16-5

VSX_DEBUG_FLAGS=xxxxx, 16-5

W

weighted allocator, 25-9

weighting, adjusting between free space and
location, 25-9

writeconf, 4-13

