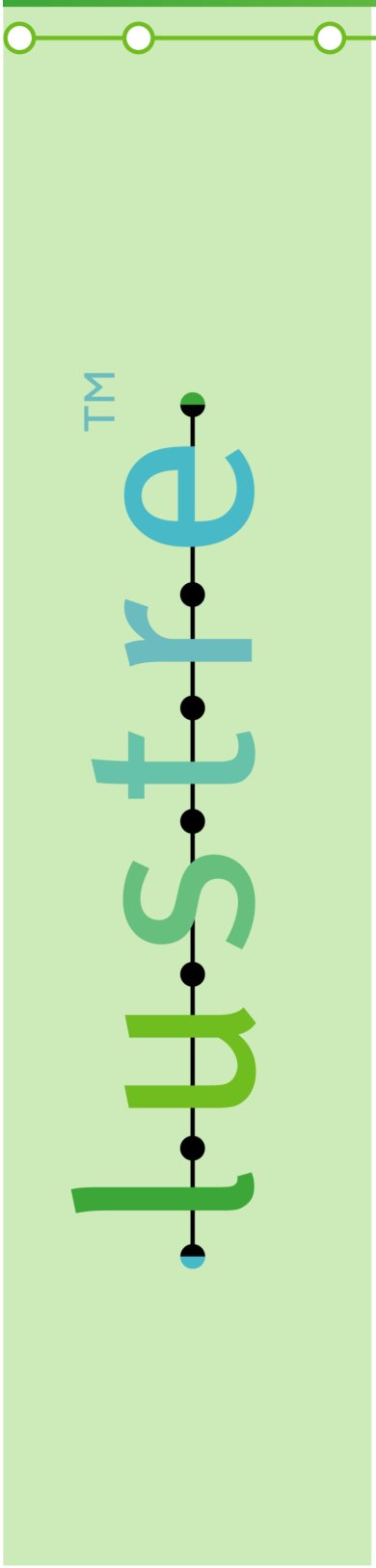

Lustre® 1.6.x Operations Manual

Version 1.6.x.1-man-v1

CFS Cluster File Systems, Inc™

Lustre 1.6.x Operations Manual

Version 1.6.x.1-man-v1 (03/05/2007)

This publication is intended to help Cluster File Systems, Inc. (CFS) Customers and Partners who are involved in installing, configuring, and administering Lustre.

The information contained in this document has not been submitted to any formal CFS test and is distributed AS IS. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by CFS for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Comments may be addressed to:

Cluster File Systems, Inc.

Suite E104 - 288

4800 Baseline Road

Boulder CO 80303

TABLE OF CONTENTS

Conventions for Command Syntax

All the commands in this manual appear in green color of the font Courier New (point size 9) with the sign '| $' in the beginning. The other conventions are described below:

- Vertical Bar: '|' : To indicate alternative, mutually exclusive elements
- Square Brackets: '[ ]' : To indicate optional elements
- Braces: '{ }' : To indicate that a choice is required by the user
- Braces within brackets: '[{ }]' : To indicate that a choice is required within an optional element
- Backslash: ' \' : To indicate that the command line is continued on the next line
- **Boldface**: To indicate that the word is to be entered literally as shown
- *Italics*: To indicate a variable or argument to be replaced by the user with an actual value

PART I. ARCHITECTURE

CHAPTER I – 1. A CLUSTER WITH LUSTRE

1.1 What is Lustre?

Lustre is a high-performance, multi-network, fault-tolerant, POSIX-compliant network file system for Linux clusters.

Key features of Lustre –

- Ability to run over a large range of network fabrics
- Fine-grained locking for efficient concurrent file access
- Failover ability to reconstruct 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.

1.2 The Software

The Lustre software consists of three interacting software areas:

- **A patched Linux kernel**

  Lustre requires significant changes from the standard Linux kernel to facilitate some of its performance improvements. The changes are distributed in the form of patches against specific kernels. Some specific pre-patched kernels are also available from our download site. Additionally, the Lustre client (but not Lustre servers) can be run on certain unmodified kernels (also known as "patchless").

- **The Lustre modules**

  Lustre kernel modules provide the server and client capabilities for the file system.

- **Userspace utilities**

  A few userspace utilities are required for configuration and startup/shutdown of Lustre servers and clients.

1.3 Lustre Components

A Lustre file system consists of four major components:

- Management Server
- Meta Data Target (MDT)
- Object Storage Targets (OSTs)
- Lustre Clients

Lustre clients provide remote access to the Lustre file system. The file system is served jointly by the Object Storage Targets (OSTs) for file contents and the Meta Data Target (MDT) for file meta data (directory structure, file size, and so on). A single Lustre file system may have multiple OSTs, each serving a subset of the file data. Note that there is not necessarily a 1:1 correspondence between a file and an OST; a file may be spread over many OSTs in order to optimize performance. Each of the OSTs and the MDT may have a failover partner to provide access to the back-end storage if the server node fails.

Figure 1.1.1: A Lustre Cluster shows the expected interactions between the servers and clients of the Lustre file system.

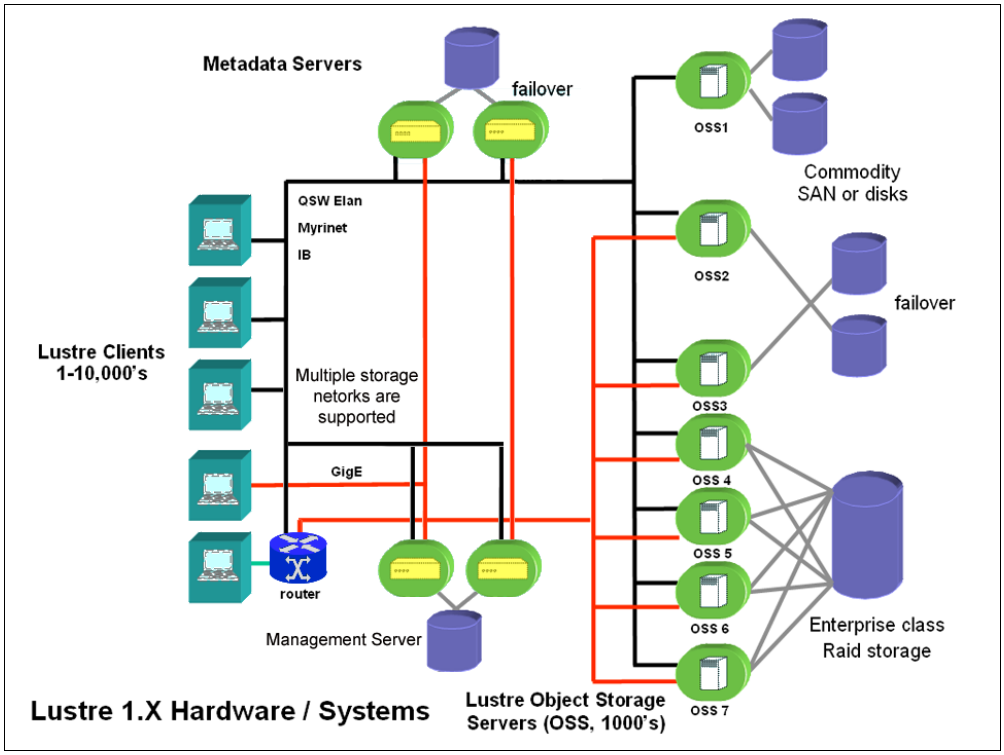

Figure 1.1.1: A Lustre Cluster

The MDT, OSTs and 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 The Management Server

The ManaGement Server (MGS) defines the configuration information for all of the Lustre file systems at a site. Each Lustre Target (below) contacts the MGS to provide information, and Lustre clients contact the MGS to retrieve the 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 to allow the MGS to share a disk ("co-locate") with a single MDT. The MGS is not "part" of an individual file system through itself. It provides configuration mechanisms to the other components.

1.3.2 The Meta Data Target

The Meta Data Target (MDT) provides back-end storage for the meta data information for a single file system. The Meta Data Server (MDS) provides the network request handling for one or more local MDTs. (Due to historical reasons, the term “MDS” has been traditionally used to mean both the MDS and a single MDT. This (and future) version of the manual will try to use the more specific meanings.)

The meta data 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 The Object Storage Targets

An Object Storage Target provides back-end storage for file object data, which are effectively chunks of user files. There are typically multiple OSTs providing access to different chunks. (The MDT keeps track of which chunks are where.) 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

The Lustre clients are "users" of the file system. They are normally computation, visualization, or desktop nodes. The Lustre clients require the Lustre software to mount a Lustre file system - 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: Meta Data 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, these provide transparent access to the file system.

All clients which mount the file system will 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 are reading 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 the Lustre clients.

1.3.5 Lustre Networking

Servers and clients communicate with each other over a custom networking API called LNET. LNET inter-operates with a variety of network transports through Network Abstraction Layers (NAL).

This API provides the delivery and event generation in connection with network messages. It also provides advanced capabilities such as using Remote DMA (RDMA) if the underlying network transport layer supports this, and autonomous routing between different network transports on different nodes.

CHAPTER I – 2. UNDERSTANDING LUSTRE NETWORKING

2.1 Introduction

In a Lustre network, servers and clients communicate with each other over a custom networking API called LNET, which abstracts away all transport-specific interaction. LNET in turn operates with a variety of network transports through LNET Device drivers (LNDs).

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

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

2.2 Supported Network Types

- tcp (Ethernet)
- openib (Mellanox-Gold Infiniband)
- iib (Infinicon Infiniband)
- vib (Voltaire Infiniband)
- o2ib (OFED)
- ra (RapidArray)
- elan (Quadrics Elan)
- gm (Myrinet)

2.3 Important Terms

LND: Lustre networking device layer, a modular subcomponent of LNET that implements one of the network types. The LNDs are implemented as individual Linux modules, and typically must be compiled against the the network driver software.

Network: A group of nodes that communicate directly with each other. It 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, elan0).

NID: A Lustre networking ID. The NID uniquely identifies a Lustre network endpoint, including the node and the network type. This means there is an NID for every network a node uses.

Part II. Lustre Administration

CHAPTER II – 1. PREREQUISITES

1.1 Preparing to Install Lustre

1.1.1 How to get Lustre

The current, stable version of Lustre is available for download from the website of Cluster File Systems:

http://www.clusterfs.com/download.html

The software available for download on this website is released under the GNU General Public License. It is strongly recommended to read the complete license and release notes for this software before downloading it, if you have not done so already. The license and the release notes can also be found at the aforementioned website.

1.1.2 Supported Configurations

Cluster File Systems, Inc. supports Lustre on the configurations listed in Table1.1.1: **Supported Configurations**.

ASPECT	SUPPORT TYPE
Operating Systems:	Red Hat Enterprise Linux 3+, SuSE Linux Enterprise Server 9, 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; Myranet, Mellanox, Infiniband (Voltaire, OpenIB and Silverstrom)

Table 1.1.1: Supported Configurations

1.2 Using a Pre-packaged Lustre Release

Due to the complexity involved in building and installing Lustre, Cluster File Systems has made available several different pre-packaged releases that cover some of the most common configurations.

The pre-packaged release consists of five different RPM packages given below. Install them 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 package.
- **kernel-source-<release-ver>.rpm** – This is the Lustre patched Linux kernel source RPM. This comes with the kernel package, but is not required to build or use Lustre.
- **lustre-modules-<release-ver>.rpm** – The Lustre kernel modules for the above kernel.
- **lustre-<release-ver>.rpm** – These are the Lustre Utilities or userspace utilities for configuring and running Lustre. Use them only with the matching kernel RPM as mentioned 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 required only if you need to build your own modules (for networking, and so on) against the kernel source.

> **NOTE:** 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. Please exercise caution and back up all data before using this software.

1.2.1 Choosing a Pre-packaged Kernel

Determining the best suitable pre-packaged kernel, depends largely on the combination of hardware and software being run. CFS provides pre-packaged releases on our download Web site.

1.2.2 Lustre Tools

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

- **lctl:** A low-level configuration utility that can also be used for troubleshooting and debugging;
- **lfs:** A tool for reading/setting striping information for the cluster, as well as performing other actions specific to Lustre File Systems;
- **mount.lustre:** The Lustre specific helper for mount(8);
- **mfks.lustre:** A tool to format Lustre target disks.

1.2.3 Other Required Software

Besides the tools provided along with Lustre, Lustre also requires some separate software tools to be installed.

1.2.3.1 Core Requirements

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

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

You might have to install it with rpm -ivh --force to override any dependency issues of your distro.

Perl: Various userspace utilities are written in Perl. Any modern Perl should work.

build tools: If you are not installing Lustre from RPMs, you can normally build Lustre with the GCC complier. You need GCC 3.0 or later.

1.2.3.2 High Availability Software

If you plan to enable failover server functionality with Lustre (either on OSS or on MDS), a high availability software will be a necessary addition to your cluster software. One of the better known high availability software packages is Heartbeat.

Linux-HA (Heartbeat) supports redundant system with access to the Shared (Common) Storage with a dedicated connectivity; and can determine the general state of the system. (For details, see **Part II - Chapter 6. Failover**.)

1.2.3.3 Debugging Tools

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

The most useful tool in this regard is GDB, coupled with crash. Together, these tools can be used to investigate both, 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.

More information about these tools can be found at the following locations:

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/

1.3 Environment Requirements

1.3.1 SSH Access

It is not strictly required, but in many cases it is very helpful to have remote ssh access to all the nodes in a cluster. Some of the Lustre configuration and monitoring scripts depend on ssh (or pdsh) access; although none of these are required for running Lustre.

1.3.2 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 source of headache as it will be very difficult to debug any multi-node issue, or otherwise correlate the logs. For this reason, CFS recommends that the machine clocks should be kept in sync as much as possible. The standard way to accomplish this is by using the Network Time Protocol, or NTP. All the machines in your cluster should synchronize their time from a local time server (or servers) at a suitable time interval.

More information about ntp can be found at:

http://www.ntp.org/

1.3.3 Universal UID/GID

In order to maintain uniform file access permissions on all the nodes of your cluster, the same user (UID) and group (GID) IDs should be used on all the clients. Pretty much like any cluster usage, Lustre uses the common UID/GID on all the cluster nodes.

1.3.4 Proper Kernel I/O Elevator

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, as follows:

- 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

Please note that the above is just our best advice, and we strongly suggest that local testing is the best way to ensure high performance with Lustre. Also note that most distributions ship with either “cfq” or “as” configured as the default scheduler, and thus choosing an alternate scheduler is an absolutely necessary step in configuring Lustre for the best performance. The “cfq” and “as” schedulers should never be used for server platform.

Please see the following resources for more in-depth discussion on choosing an I/O scheduler algorithm for Linux:

- http://www.redhat.com/magazine/008jun05/features/schedulers
- http://www.novell.com/brainshare/europe/05_presentations/tut303.pdf
- http://kerneltrap.org/node/3851

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 (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), 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 will contain 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
```

The other schedulers (anticipatory and cfq) are better suited for desktop use.

CHAPTER II – 2. LUSTRE INSTALLATION

2.1 Installing Lustre

Follow the steps outlined below to install Lustre:

1. Install the Linux base OS as per your requirements along with the prerequisites like GCC and Perl (as mentioned in **Part II – Chapter 1. Prerequisites**).
2. Install the RPMs as described in section **1.2 Using a Pre-packaged Lustre Release**, in **Part II – Chapter 1. Prerequisites**. The preferred installation order is:
   i. the Lustre patched version of the linux kernel (kernel-*)
   ii. the Lustre modules for that kernel (lustre-modules-*)
   iii. the Lustre user space 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 **5.1.1** and **5.2.2 Modprobe.conf** in **Part II – Chapter 5. More Complicated Configurations**.)

TIP:
When installing Lustre with InfiniBand you need to keep the ibhost, kernel and Lustre all on the same revision. Follow these steps to achieve 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.

2.1.1 MountConf

MountConf is shorthand for Mount Configuration. Lustre cluster configuration is accomplished by the mkfs.lustre and mount commands only. There are no more lconf, lmc, xml as in previos versions of Lustre. The MountConf system is one of the important new features of Lustre 1.6.0.

MountConf involves userspace utilites (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 file systems running at a site. There should be one MGS per site, not one MGS per file system. 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 file system.

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 file system. (However, there should be no client I/O at addition time, also kknown as "quiescent ost addition.")

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

i. start mgs
ii. mkfs, mount ost #1
iii. mkfs, mount mdt
iv. mkfs, mount ost #2
v. mount client
vi. mkfs, mount ost #3

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

NOTE: The MGS must be running before any new servers are added to a filesystem. After the first time the servers start, they cache a local copy of their startup logs so that 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 Lustre client.

2.2 Quick Configuration of Lustre

As we have already discussed, Lustre consists of four types of subsystems – a Management Server (MGS), a Meta Data Target (MDT), Object Storage Targets (OSTs) and clients. All of these can co-exist on a single system or can run on different systems. The object storage servers and meta data server 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. CFS includes some sample scripts in the directory where Lustre is installed. The scripts are located in the *lustre/tests* subdirectory if you have installed the source code. These scripts enable quick setup of some simple, standard configurations.

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

2.2.1 Simple Configurations

2.2.1.1 Module Setup

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

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
```

Now 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
```

2.2.1.2 Making and Starting a File System

Starting Lustre on MGS and MDT Node “mds16”

First create an MDT for the file system "spfs" that uses the disk /dev/sda. This MDT will also act 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 blocks     0
         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,mballoc
Parameters: mgsnode=192.168.16.21@tcp


device size = 4096MB
formatting backing filesystem ldiskfs on /dev/sda
        target name  spfs-OSTffff
        4k blocks     0
        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 client node

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

The MGS and the MDT can be run on separate devices instead. 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 given and the client will choose 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
```

2.2.1.3 File System Name

File system name is limited to 8 characters. CFS has encoded the file system and target information in the disk label, so that you can mount by label. This allows syste administrators to move disks around without worrying about issues like SCSI disk reordering or getting the /dev/device wrong for a shared target. CFS will soon make this as failsafe as possible. The current Linux disk labels are limited to 16 characters. We reserve 8 of those characters for identifying the target within the file system, leaving 8 characters for the file system name:

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

An example mount-by-label:

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

One mitigating factor might be that although the FS name is internally limited to 8 characters, you can mount the clients at any mountpoint, so the file system users would never be subject to short names:

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

2.2.1.4 Starting a Server Automatically

As 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 HA package manage when to mount the device. If you are not using failover, you should still insure that networking has been started before mounting a Lustre server. RedHat, SuSe, Debian (maybe others) use the "_netdev" flag to insure that these disks are mounted after the network is up.

Note that 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 will end in FFFF, meaning that it has yet to be assigned. The assignment will take place when the server is first started, and the disk label will be updated.

2.2.1.5 Stopping a Server

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

The '-f' flag means "force", force the server to stop WITHOUT RECOVERY (equivalent to the old lconf –force). Without the '-f' flag, "failover" is implied, meaning the next time the server is started it will go through the recovery procedure (equivalent to the old lconf --failover).

> **NOTE:** If you are using loopback devices, use the '-d' flag. This flag cleans up loop devices and can always safely be specified.

2.2.2 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, all of its NIDs are generally required to be listed (delimited by commas ','), so that 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), you can execute the following command –

```
lctl list_nids
```

2.2.2.1 Failover

This example has a combined MGS/MDT failover pair on uml1 and uml2, and a OST failover pair on uml3 and uml4. uml1 and uml2 have corresponding elan addresses as well.

```
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 (uml2,2@elan) means that these two NIDs refer to the same host, and that Lustre just needs to choose the "best" one of the two for communication. Colon-separation (uml1:uml2) means that the two NIDs refer to two different hosts, and should be treated as failover locations (Lustre will try the first one, and if that fails, it will try the second one.)

2.2.2.2 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.

2.2.2.3 Without Lustre Service

Start only the MGS or MGC, and not the target server (for instance, 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
```

2.2.2.4 Failout

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

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

2.2.2.5 Running Multiple Lustres

The default file system name created by mkfs.lustre is "lustre." Specify "mkfs.lustre --fsname=foo" for a different fs name. The MDT, OSTs and clients that comprise a single file system must share the same name, for instance:

```
foo-MDT0000

foo-OST0000

foo-OST0001

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

The maximum length of the file system name is 8 characters.

Note that the MGS is universal. In the sense, there is only one MGS per installation, not one per file system. So an installation with two file systems could look like:

```
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
```

2.2.3 Other Configuration Tasks

2.2.3.1 Removing an OST Permanently

For Lustre 1.6, an OST can be permanently removed from a file system. Note that any files that have stripes on the removed OST will henceforth return EIO.

```
$ mgs> lctl conf_param testfs-OST0001.osc.active=0
```

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

To restore the OST, make sure it is running, and then run the following command:

```
$ mgs> lctl conf_param testfs-OST0001.osc.active=1
```

2.2.3.2 Writeconf

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

- you have got the config logs into a state where the file system cannot start; or

- you are changing the NIDs of one of the servers.

Follow the writeconf procedure given below:

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

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

3. Remount all servers, mounting the ***MDT first***.

2.2.3.3 Changing a Server NID

1. Update the LNET configuration in /etc/modprobe.conf so that lctl list_nids is correct.
2. Regenerate the configuration logs for every affected file system using the --writeconf flag to tunefs.lustre, as shown in the 2nd step of the section **2.2.2.4 Writeconf**.
3. If the MGS NID is also changing, then communicate the new MGS location to each server by using

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

2.2.3.4 Abort Recovery

Abort the recovery process when starting a target:

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

NOTE: The recovery process will currently get blocked until all OSTs are available.

2.3 Building from Source

2.3.1 Building Your Own Kernel

In the case that the hardware is not standard or CFS support have asked that you apply a patch, Lustre will require some changes to the core Linux kernel. These changes are organized in a set of patches in the kernel_patches directory of the Lustre CVS repository. If you are building your kernel from the source you will need to apply the appropriate patches.

Managing patches for the kernels is a very involved process given that most patches are intended to work with several kernels. To facilitate support, CFS maintains the tested version on the FTP site as some versions may not work properly with the patches from CFS. We recommend you use the Quilt package developed by Andreas Gruenbacher as it simplifies the process considerably. Patch management with Quilt works as follows:

- a series file lists a collection of patches
- the patches in a series form a stack
- using Quilt you then push and pop the patches
- you then edit and refresh (update) the patches in the stack that is being managed with Quilt
- you can then revert inadvertent changes and fork or clone the patches and conveniently show the difference in work, before and after.

2.3.1.1 Patch Series Selection

Depending on the kernel being used, a different series of patches needs to be applied. CFS maintains a collection of different patch series files for the various supported kernels in the directory lustre/kernel_patches/series/. This directory is in the Lustre tarball distributed by CFS.

For instance, the file lustre/kernel_patches/series/rh-2.4.20 lists all the patches that should be applied to a 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 always be found in the file lustre/kernel_patches/which_patch.

2.3.1.2 Installing Quilt

A variety of quilt packages (RPMs, SRPMs and tarballs) are available from various sources. We recommend you use a recent version of quilt, at least version 0.29. If

possible, use a quilt package from your distribution vendor. If this is not possible, you may download a package from the ftp site of Cluster File Systems:

ftp://ftp.clusterfs.com/pub/quilt/

If you cannot find an appropriate quilt package or cannot fulfill it's dependencies, we suggest building quilt from the tarball. You can download the tarball from the main quilt website:

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

2.3.1.3 Preparing the Kernel Tree Using Quilt

After acquiring the Lustre source (CVS or tarball) and choosing a series file to match your kernel sources you must also choose a kernel config file. The supported kernel ".config" files are in the folder lustre/kernel_patches/kernel_configs, and are named in such a way as to indicate which kernel and architecture they are meant for. For example, kernel-2.6.9-2.6-rhel4-x86_64-smp.config is a config file for the 2.6.9 kernel shipped with RHEL 4 suitable for x86_64 SMP systems.

Next unpack the appropriate kernel source tree. For the purposes of illustration, this documentation assumes that the resulting source tree is in /tmp/kernels/linux-2.6.9, we will refer to this as the destination tree.

You are now ready to use Quilt to manage the patching process for your kernel. The following set of commands will setup 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 .
```

You can now have quilt apply all the patches in the chosen series to your kernel sources by using the set of commands given below.

```
$ 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 now 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.

2.3.2 Building Lustre

The Lustre source can be obtained by registering on the site:

http://www.clusterfs.com/download.html

Once you register you will receive an email with the link for download.

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

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

- Example of **binary packages** for 1.5.97:
- kernel-lustre-smp-2.6.9-42.0.3.EL_lustre.1.5.97.i686.rpm will contain patched kernel
- lustre-1.5.97-2.6.9_42.0.3.EL_lustre.1.5.97smp.i686.rpm will contain Lustre user space files and utilities
- lustre-modules-1.5.97-2.6.9_42.0.3.EL_lustre.1.5.97smp.i686.rpm will contain Lustre modules (kernel/fs/lustre and kernel/net/lustre).

You can install the binary packages by issuing the standard RPM commands:

```
$ 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
```

- Example of **Source packages**:
- kernel-lustre-source-2.6.9-42.0.3.EL_lustre.1.5.97.i686.rpm will contain the source for the patched kernel
- lustre-source-1.5.97-2.6.9_42.0.3.EL_lustre.1.5.97smp.i686.rpm will contain the source for Lustre modules and user space utilities.

The 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 you can build Lustre by running the sequence of commands given below.

```
$ cd <path to kernel tree>

$ cp /boot/config-'uname -r' .config

$ make oldconfig || make menuconfig
```

For 2.6 kernels

```
  $ make include/asm

  $ make include/linux/version.h

  $ make SUBDIRS=scripts
```

For 2.4 kernels

```
  $ make dep
```

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

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

$ make rpms
```

This will create 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.xxEL_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
```

cd into the kernel source directory and run

```
$ make rpm
```

This will create a kernel RPM suitable for the installation.

Example: kernel-2.6.95.0.3.EL_lustre.1.5.97custom-1.i386.rpm

2.3.2.1 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 listing of the supported features and packages can always be obtained by issuing the command "./configure –help" in your 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.

2.3.2.2 Liblustre

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

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

> **NOTE:** 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 checkout the b_lustre branch from the libsysio CVS repository. This gives the version of libsysio compatible with Lustre. Once checked out, the steps listed below will build libsysio.

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

Once libsysio is built, you can build liblustre using the following commands.

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

2.3.2.3 Compiler Choice

The compiler must be greater than GCC version 3.3.4. GCC v4.0 is not currently supported. GCC v3.3.4 has been used to successfully compile all of the pre-packaged releases made available by CFS, and as such is the only compiler that is officially supported. Your mileage may vary with other compilers, or even with other versions of GCC.

> **NOTE:** GCC v3.3.4 was used to build 2.6 series kernels.

CHAPTER II – 3. CONFIGURING THE LUSTRE NETWORK

3.1 Designing Your Network

Before configuration can take place, a clear understanding of your Lustre network topologies is essential.

3.1.1 Identify all Lustre Networks

A network is a group of nodes that communicate directly with each other. As mentioned previously, Lustre supports a variety of network types and hardware, including TCP/IP, Elan, varieties of Infiniband and others. The normal rules for specifying networks apply, for example, two TCP networks on two different subnets would be considered two different Lustre networks. For example, tcp0 and tcp1.

3.1.2 Identify nodes which will 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 (For example, TCP to Elan) or across different topologies (For example, bridging two Infiniband or TCP/IP networks).

3.1.3 Identify any network interfaces that should be included/excluded from Lustre networking

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

3.1.4 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 distro). 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. See the ip2nets option below for more information.

LibLustre users should set the accept=all parameter. See the section **3.2.1 Module Parameters** for details.

3.1.5 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, clients should use the NID appropriate for its local networks. If unsure, there is a **lctl** command that can help. On the MDS,

```
lctl list_nids
```

will display the server's NIDs. On a client,

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

will display the closest NID for that client. So from a client with SSH access to the MDS,

```
mds_nids=`ssh the_mds lctl list_nids`

lctl which_nid $mds_nids
```

will in general be the correct NID to use for the MDS in the mount command.

3.2 Configuring Your Network

> **NOTE:** We recommend using 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.

3.2.1 Module Parameters

LNET network hardware and routing are now 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 instance:

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

specifies that this node should use all available TCP and elan interfaces.

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's name.

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

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

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

But the additional quotes may confuse some distributions. Check for messages such as:

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

After modprobe LNET, the additional single quotes should be removed from modprobe.conf in this case.

Additionally, the message "refusing connection - no matching NID" generally points to an error in the LNET module configuration.

> **NOTE:** By default, Lustre will ignore the loopback (lo0) interface. Lustre will not ignore IP addresses aliased to the loopback. Specify all Lustre networks in this case.

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.

Please note that 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 RPC replies via a router that the liblustre client has not already connected to, these RPC replies will be 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.

SilverStorm InfiniBand Options -

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

options lnet networks="iib3(2)"

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

3.2.2 Module Parameters – Routing

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

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

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 section **5.1.1 Modprobe.conf** of **Part II – Chapter 5. More Complicated Configurations**.)

And here is a very complex routed configuration with Voltaire Infiniband and Myranet (GM) systems, with four systems configured as routers:

```
options lnet\
     ip2nets="gm  10.10.3.*              # aa*-i0;\
              vib  10.10.131.[11-18]     # aa[11-18]-ipoib0;\
              vib  10.10.132.*           # cc*-ipoib0;"\
     routes="gm   10.10.131.[11-18]@vib  # vib->gm via aa[11-13];\
             vib   0xdd7f813b@gm         # gm->vib via aa11;\
             vib   0xdd7f81c7@gm         # gm->vib via aa12;\
             vib   0xdd7f81c2@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. CFS is 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 parameter **live_router_check_interval** specifies a time interval in seconds after which the router checker will ping the live routers.

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

You can set the timeout for the router checker to check the live or dead routers by setting the parameter **router_ping_timeout**. 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 believes the router is 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 for which it is disabled.

The initial time to disable a router should be 1 minute (enough to plug in a cable after removing it usually). If the router is administratively marked as "up", the router checker clears the timeout. When a route is disabled, the (possibly new) "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 is 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. You should set the timeout to 0, so that future errors will 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.

3.2.3 Downed Routers

There are two mechanisms to update 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. This is off by default. 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 will 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 will ignore all such peer-down notifications.

Some key differences in both the mechanisms:

1. The router pinger only checks routers for their health, while LNDs can notice all dead peers irrespective of whether they are a router or not.

2. The router pinger checks the router health actively by sending pings, but LNDs can only notice a dead peer when there is network traffic going on.

3. The router pinger can bring a router from alive to dead or vice versa, but LNDs can only bring a peer down.

3.3 Starting and Stopping LNET

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

3.3.1 Starting LNET

The command to start the lnet is -

```
$ modeprob lnet
$ lctl network up
```

To see the list of local nids -

```
$ lctl list_nids
```

This will tell you if your local node's networks are set up correctly. If not, see modules.conf "networks=" line and insure the network layer modules are correctly installed and configured.

To get the best remote nid -

```
$ lctl which_nid
```

This will take the "best" nid from a list of the nids of a remote host. The "best" nid is the one the local node will use when trying to communicate with the remote node.

3.3.1.1 Starting Clients

TCP client:

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

Elan client:

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

3.3.2 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. It is to stop the LNET network by using the following 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. So it is advised to be certain that the Lustre network and Lustre are

stopped prior to module unloading, and to be extremely careful when using *rmmod -f*.

To unconfigure LCTL network, following command can be used:

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

lconf --cleanup
```

This command will do the Lustre and LNET cleanup automatically in cases where lconf was used to start the services.

TIP:
To remove all the Lustre modules:
$ lctl modules | awk '{print $2}' | xargs rmmod

CHAPTER II – 4. CONFIGURING LUSTRE - EXAMPLES

4.1 Simple TCP Network

Below are some examples of Lustre configurations on simple TCP network.

4.1.1 Lustre with Combined MGS/MDT

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

4.1.1.1 Installation Summary

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

4.1.1.2 Configuration Generation and Application

- Install the Lustre RPMS as per the section **2.1 Installing Lustre** of **Part II – Chapter 2. Lustre Installation** on all the nodes that are going to be a part of the Lustre file system. Boot the nodes in Lustre kernel including the clients
- Change modprobe.conf by adding the following line to it

```
options lnet networks=tcp
```

- Start Lustre on MGS and MDT Node

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

- Make a mount point on MDT/MGS for the file system and mount it

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

- Start Lustre on all the four OSTs

  On OST-0

```
mkfs.lustre --fsname datafs --ost0 --mgsnode=mds16@tcp0 /dev/sda
```

  On OST-1

```
mkfs.lustre --fsname datafs --ost1 --mgsnode=mds16@tcp1 /dev/sdd
```

  On OST-2

```
mkfs.lustre --fsname datafs --ost2 --mgsnode=mds16@tcp2 /dev/sda1
```

  On OST-3

```
mkfs.lustre --fsname datafs --ost3 --mgsnode=mds16@tcp3 /dev/sdb
```

- Make a mount point on all the OSTs for the file system and mount it

  On OST-0

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

  On OST-1

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

  On OST-2

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

  On OST-3

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

- On the client

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

4.1.2 Lustre with Separate MGS and MDT

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

4.1.2.1 Installation Summary

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

4.1.2.2 Configuration Generation and Application

- Install the Lustre RPMs as per the section **2.1 Installing Lustre** of **Part II – Chapter 2. Lustre Installation** on all the nodes that are going to be a part of the Lustre file system. Boot the nodes in Lustre kernel including the clients
- Change the modprobe.conf by adding the following line to it

```
options lnet networks=tcp
```

- Start Lustre on the MGS node

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

- Make a mount point on MGS for the file system and mount it

```
$ mkdir -p /mnt/mgs

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

- Start Lustre on the MDT node

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

- Make a mount point on MDT/MGS for the file system and mount it

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

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

- Start Lustre on all the four OSTs

  On OST-0

```
mkfs.lustre --fsname datafs --ost0 --mgsnode=mds16@tcp0 /dev/sda
```

  On OST-1

```
mkfs.lustre --fsname datafs --ost1 --mgsnode=mds16@tcp1 /dev/sdd
```

  On OST-2

```
mkfs.lustre --fsname datafs --ost2 --mgsnode=mds16@tcp2 /dev/sda1
```

  On OST-3

```
mkfs.lustre --fsname datafs --ost3 --mgsnode=mds16@tcp3 /dev/sdb
```

- Make a mount point on all the OSTs for the file system and mount it

  On OST-0

```
$ mkdir -p /mnt/data/ost0

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

  On OST-1

```
$ mkdir -p /mnt/data/ost1

$ mount -t lustre /dev/sdd /mnt/data/ost1
```

  On OST-2

```
$ mkdir -p /mnt/data/ost2

$ mount -t lustre /dev/sda1 /mnt/data/ost2
```

  On OST-3

```
$ mkdir -p /mnt/data/ost3

$ mount -t lustre /dev/sdb /mnt/data/ost3
```

- On the client

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

CHAPTER II – 5. MORE COMPLICATED CONFIGURATIONS

5.1 Multihomed Servers

Servers megan and oscar each have three tcp NICs (eth0, eth1, and eth2) and an elan NIC. eth2 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.

5.1.1 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 of these options 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 kiiblnd ipif_basename=ib0
```

> **NOTE:** In case of TCP-only clients, all the available IP interfaces will be 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 NID.

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 this option (ip2nets) are used **only** 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 used by the local node only 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.

5.1.3 Start Servers

For the combined MGS/MDT with TCP Network

```
$ 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

```
$ 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

```
$ 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

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

5.1.4 Start Clients

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

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

You can start the Elan clients with:

```
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, tcp client could use uml1@tcp0, and elan client could use 1@elan). Alternatively, a list of all MGS nids can be given, and the client will choose the correct one.
For example: $ mount -t lustre mgs16@tcp0,1@elan:/testfs /mnt/testfs

5.2 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.

5.2.1 Modprobe.conf

Modprobe.conf is identical on all nodes:

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

5.2.3 Start servers

router1

```
modprobe lnet

lctl network configure
```

megan and oscar:

```
FIXME
```

5.2.4 Start clients

tcp client:

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

elan client:

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

CHAPTER II – 6. FAILOVER

6.1 What is Failover?

We say a computer system is **Highly Available** when the services it provides are available with minimum downtime. Even in case of failure conditions such as loss of a server, or network or software fault, the services being provided remain unaffected for the user. We generally measure availability by the percentage of time we require the system to be available.

Availability is accomplished by providing replicated hardware and/or software, so that failure of any system will be covered by a paired system. What we call “failover” is a method of automatically switching an application and it's supporting resources to a standby server when the primary system fails or the service is temporarily shut down for maintenance. 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.

Lustre provides a file system resource. The Lustre file system supports failover at the server level. Lustre does not provide the tool set for the system-level components necessary for a complete failover solution (node failure detection, power control, and so on), as this functionality has been available for some time from third party tools. CFS does provide the necessary scripts to interact with these packages, and exposes health information for system monitoring. The recommended choice is the Heartbeat package from linux-ha.org. Lustre will work with any HA software that supports resource (I/O) fencing. The Heartbeat software is responsible for detecting failure of the primary server node and controlling the failover.

The hardware setup requires a pair of servers with a shared connection to a physical storage (like SAN, NAS, hardware RAID, SCSI, Fiber Channel). The method of sharing the 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 high available Lustre system, one needs a power management software and HA software, which must provide the following -

A) -- Resource fencing - Physical storage must be protected from simultaneous access by two nodes

B) -- Resource control - Starting and stopping the Lustre processes as a part of failover, maintaining the cluster state, and so on

C) -- 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 the 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. CFS provides /etc/init.d/lustre for this purpose.

Servers providing Lustre resources are configured in primary/secondary pairs for the purpose of failover. A system administrator can failover manually with Iconf. 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 the node off is known as a **hard** failover.

To automate failover with Lustre, one needs a power management software, remote control power equipment, and HA software.

6.1.1 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, is a tool for manipulating remote power control (RPC) devices from a central location. Several RPC varieties are supported natively by PowerMan.

The latest version is available on

http://www.llnl.gov/linux/powerman/

6.1.2 Power Equipment

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

6.1.3 Heartbeat

The heartbeat program is one of the core components of the Linux-HA (High-Availability Linux) project. Heartbeat is highly portable, and runs on every known Linux platform, and also on FreeBSD and Solaris.

For more information, see:

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

For download, go to:

http://linux-ha.org/download

CFS 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. The linux-ha web site contains a great deal of information. We recommend it as a resource.

6.1.3.1 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 and a *passive* node is idle, standing by to take over in the event of a failure. In the example case of 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 will serve 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 would take over for the other.

To configure this with respect to the shared disk, the shared disk would need to provide multiple partitions, and each of the OSTs would be 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.

6.2 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, get 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.

6.3 MDS Failover Review

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

6.4 Configuring MDS and OSTs for Failover

6.4.1 Starting / Stopping a Resource

You can start a resource with “mount” command and stop it with “umount” command. For more details, see the section **2.2.1.6 Stopping a Server** in **Part II – Chapter 2. Lustre Installation**.

6.4.2 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 is to demonstrate the /dev/ identify can differ between nodes, but both devices must map to the same physical LUN.

For a failover example, see the section **2.2.1.1 Single System Test with llmount.sh Script** in **Part II – Chapter 2. Lustre Installation**.

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.

6.4.3 Hardware Configurations

6.4.3.1 Hardware Preconditions

1. 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).
2. 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. Heartbeat will assume that the non-primary node is secondary for that resource.
3. 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.

4. 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.
5. 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. Typically, this would be a firewall router, or another very reliable network endpoint external to the cluster.

6.5 Instructions for Failover Setup with Heartbeat Version1

6.5.1 Software Installations

1. Install Lustre as described in Chapter II – 2. **Lustre Installation**.

2. Install RPMs required for configuring Heartbeat

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

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 the location given below -

http://linux-ha.org/download/index.html#1.2.3

3. Install 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

6.5.2.2 Lustre Configuration

- Create the directory **/etc/lustre**
- Verify that **/etc/init.d/lustre** exists
- Note the names of your OST and MDS resources

- Decide which node owns each resource

6.5.2.3 Heartbeat Configuration

A. Basic Configuration - no STONITH

The linux-ha web site has several guides covering basic setup and initial testing of Heartbeat, we advise reading them.

1. It is good to configure and test the Heartbeat setup before adding STONITH.

Let us assume 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 the nodes are pinging to a remote host – 192.168.0.3 for health.

a. Create /etc/ha.d/ha.cf

- This file must be identical on both the nodes
- Follow the order of the directives as it matters
- See sample ha.cf file in the section **6.5.5.3 ha.cf** of this chapter

b. Create /etc/ha.d/haresources

- This file must be identical on both the nodes
- It specifies a virtual IP address, and a service
- See sample in the section **6.5.5.4 haresources** of this chapter
- The virtual IP address should be a subnet matching a physical Ethernet. Failure to do so will result in error messages, but these errors will not be fatal.

c. Create /etc/ha.d/authkeys

- Copy example from /usr/share/doc/heartbeat-<version>
- chmod the file '0600' – heartbeat will not start if the permissions on this file are incorrect.

d. Execute the following commands to create symlinks between /etc/init.d/lustre and /etc/ha.d/resource.d/<lustre service name>

```
$ ln -s /etc/init.d/lustre /etc/ha.d/resource.d/ost1

$ ln -s /etc/init.d/lustre /etc/ha.d/resource.d/ost2
```

e. Restart heartbeat

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

```
Sep  7 10:42:40 d1_q_0 heartbeat: info: Running \
/etc/ha.d/resource.d/ost1  start
```

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

- If you configure two nodes as primary for one resource, you will see both nodes attempt to start it. This is very bad. Shutdown immediately and correct your

haresources 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 will produce huge volumes of data.

f. 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.

B. Basic Configuration - Adding 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 will require 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:

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

```
--------/etc/ha.d/ha.cf-------------------

stonith <type> <config file>
```

b. 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 un-used parameter.

To get the proper syntax:

```
$ stonith -L
```

The above command lists supported models.

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

The above command lists required parameters, and specifies config file name.

You should attempt a test with

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

This will also give data on what is required. You will be able to test by using a real host name. 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 shutdown properly. This means many tests will 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 (section **6.5.5.2 lustre_fail.alert**) 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:

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

This script will force a boot

```
$ '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
```

6.5.3 Mon (Status Monitor)

- Mon requires two scripts:
    i. A monitor script, which checks a resource for health
    ii. An alert script, which is 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 will trigger 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.

6.5.3.1 Mon Setup and Configuration

A. Install Prerequisites for Mon

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

Heartbeat monitors the health of the node. Adding Mon to the setup allows us to monitor application health, the application in this case being Lustre.

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, when installing Perl we recommend using CPAN. The packages are also available as tarballs (see cpan.org).

B. Install Mon

After installing the Perl packages, get the Mon tarball from:

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

- Untar the tarball
- Copy the Mon program to a location on the root path

  (/usr/lib/mon/mon is default)

- Install the *moncmd* program
- For this setup, CFS has altered the Mon startup a bit (see the section **6.5.5.10 S99mon.patch**). You must patch the S99mon script, and install the result as /etc/init.d/mon – set this routine to start at boot, prior to heartbeat startup

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

- Verify that the path for moncmd in the init script matches where you installed moncmd (/usr/local/bin/moncmd is the default).
- 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

- Create the /etc/mon/auth.cf file - allow everything in the *command section* change AUTH_ANY to *all*.
- 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 CFS user has provided a shell script that will generate a mon.cf file. It is provided in the section **6.5.5.7 mon.cf**.

- Copy the supplied trap generator script (mon.trap) to a proper location (/usr/local/lib/mon/)

  a. This Perl script is based on a script found on the Mon mailing list. Other scripts are also available there

- 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

  c. This script is based on /etc/init.d/lustre

- Copy the provided Lustre alert script to the mon alert directory

  (/usr/local/lib/mon/alert.d)

  a. Verify the name matches script specified in /etc/mon/mon.cf

  b. This is a stock script from the mon package

  c. For Lustre failover sequence you are free to choose another method of triggering the transition

  - The script will _not STONITH the node
  - You should edit the script to provide hard node power off or reboot if needed

C. Add Mon to the heartbeat configuration.

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

6.6 Instructions for Failover Setup with Heartbeat Version2

6.6.1 Software Installations

1. Install Lustre as described in **Part II – Chapter 2. Lustre Installation**.

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 location given below:

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

3. Install Prerequisites.

To install Heartbeat 2.0.4-1, you require:

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

6.6.2 Hardware Configurations

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.

6.6.2.1 Hardware Preconditions

The basic cluster assumptions are the same as those for Heartbeat v1. We are re-iterating the preconditions for the sake of clarity.

1. 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).
2. 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.
3. 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.
4. 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.
5. 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.

6.6.2.2 Lustre Configuration

- Lustre configuration is identical to the V1 case.

6.6.2.3 Heartbeat Configuration

See the link below for thorough details on all the configuration options:

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

As mentioned earlier, you can run Heartbeat v2 with v1 configuration. To convert from v1 configuration to v2, use the **haresources2cib.py** script, typically found in **/usr/lib/heartbeat**. If you are starting with v2, we recommend creating 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**. Further information on CiB can be found at:

http://linux-ha.org/clusterinformationbase/userguide

A. 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.

B. 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

a. 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**. This gives us more flexibility for script parameters. Scripts found in /etc/ha.d/resource.d are considered to be of type **heartbeat** and have more restrictions.

b. Create the basic ha.cf and haresources files

- haresources no longer requires the dummy virtual IP address.

Example of /etc/ha.d/haresouces

```
d1_q_0 ost-alpha
```

```
d2_q_0 ost-beta
```

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
```

c. Examine the cib.xml file

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

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

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

- Start Heartbeat

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

C. Basic Configuration – Adding STONITH

As per **B. Basic Configuration – Adding STONITH** in the section **6.5.2.3 Heartbeat Configuration**. The best way to do this is to add the STONITH options to ha.cf and run the conversion script. A sample example is in the section **6.6.4.1 ha.cf**. See http://linux-ha.org/externalstonithplugins for more information.

6.6.3 Operation

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

A. Initial startup

- Stop heartbeat if running
- If this is a new Lustre file system:

```
lconf --reformat /etc/lustre/config.xml (both nodes)

lconf --cleanup /etc/lustre.config.xml (both nodes)
```

- If this is a new Lustre configuration, remember to lconf

```
write_conf on the MDS
```

- /etc/init.d/heartbeat start on one node
- tail -f /var/log/ha-log to see progress
- After initdead, this node should start all Lustre objects

- /etc/init.d/heartbeat start on second node
- After heartbeat is up on both the nodes, failback the resources to the second node. On the second node, run:

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

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

B. Testing

- Pull power from one node
- Pull networking from one node
- After Mon is setup, pull the connection between the OST and the backend storage

C. Failback

In normal case, do the failback manually after determining that the failed node is now good. Lustre clients can work during a failback, but block momentarily.

6.7 Considerations With Failover Software and Solutions

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

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

Likewise, the Lustre *health_check* mechanism 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*.

There will indeed 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.

CHAPTER II – 7. CONFIGURING QUOTAS

7.1 Working with Quotas

Quotas allow a system administrator to limit the maximum amount of disc space a user or group can consume in a directory. Quotas are set by root, and can be set for both 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 first. If a quota for that group exists, the size of the file is counted towards that group's quota. If no quota exists for the group, the owner's user quota is checked before the file is written.

Lustre quota enforcement differs from standard Linux quota support in several ways:

- it is administered via the lfs command
- the quota is distributed (as Lustre is a distributed file system), which has several ramifications
- the quota is allocated and consumed in a quantized fashion
- the client does not set the *usrquota* or *grpquota* options to **mount**. When a quota is enabled, it is enabled for all clients of the file system and turned on automatically at mount.

7.1.1 Configuring Disk Quotas

Enabling Quotas

- If you have re-complied your Linux kernel, please be certain that CONFIG_QUOTA and CONFIG_QUOTACTL are enabled (quota is enabled in all the Linux 2.6 kernels supplied by CFS)
- FIXME add server starup instrucitons
-
- Mount the Lustre file system on the client and verify that the lquota module has loaded properly by using the lsmod command
- The mount command for Lustre no longer recognizes the *usrquota* and *grpquota* options, please remove them from your /etc/fstab if they were specified previously
- When quota is enabled on the file system, it is automatically enabled for all clients of the file system

> **NOTE:** Lustre with Linux Kernel 2.4 will not support quotas.

7.1.2 Creating Quota Files and Quota Administration

Once each quota-enabled file system is remounted, it will be capable of working with

disk quotas. However, the file system itself is not yet ready to support quotas. The next step is to run the lfs command with the *quotacheck* option:

```
#lfs quotacheck -ug /mnt/lustre
```

The quota will be turned on by default after quotacheck completes. The options that can be used are as follows:

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

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 file systems. The file system quota files must be present in the root directory of the specified file system
- quotaoff — announces to the system that the specified file systems should have all 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 1000 1100 /mnt/lustre
```

  Description: sets limits for a user "bob". The block hard limit is around 3GB and the inode hard limit is 1100. Please note: This example uses very tiny limits.

  - 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 502):

     Filesystem  blocks   quota   limit   grace    files   quota \
limit   grace

    /mnt/lustre       0  307200  309200                0    1000 \
1100

     mds-l_UUID       0       0   10240                0       0 \
200

 ost-alpha_UUID       0       0   10240

  ost-beta_UUID       0       0   10240

   ost-gam_UUID       0       0   10240
```

- Quotachown sets or changes the file owner and the group on OSTs of the specified file system.

```
$ lfs quotachown –I /mnt/lustre
```

7.1.3 Quota Allocation

The Linux kernel sets a default quota size of 1MB. Lustre handles quota allocation in a different manner. A quota must be set properly or users may experience unnecessary failures. The file system block quota is divided up among the OSTs within the file system. Each OST requests an allocation which is increased up to the quota limit. The quota allocation is then *quantized* to reduce the number of quota-related request traffic. By default, Lustre will allocate 100MB per OST. This means the minimum quota that can be assigned is 100 MB multiplied by the number of OSTs in your file system. If you attempt to assign a smaller quota, users maybe unable to create files. The default is established at file system creation time, but can be tuned via /proc values (detailed below). The inode quota is also allocated in a quantized manner on the MDS.

The *setquota* example above was run on a file system created with the following **lmc** quota options:

```
--quota quotaon=ug,bunit=10,iunit=200
```

This sets a much smaller granularity. We have specified that we will request new quota in units of 10 MB and 200 inodes respectively. If we look at the example again:

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

Disk quotas for user bob (uid 502):

     Filesystem  blocks   quota   limit   grace    files   quota \
limit   grace

    /mnt/lustre       0  307200  309200               0    1000 \
1100

     mds-l_UUID       0       0   10240               0       0 \
200

 ost-alpha_UUID       0       0   10240

  ost-beta_UUID       0       0   10240

   ost-gam_UUID       0       0   10240
```

We see that the 3GB quota requested is divided across the OSTs, with each OST having an initial allocation of 10MB blocks. The MDS line shows the initial 200 inode allocation.

It is very important to note that **the block quota is consumed per OST**. Much like free space, when the quota is consumed on one OST, clients may be unable to create files regardless of the quota available on other OSTs.

More details:

Lustre quota allocation is controlled by two values — quota_bunit_sz and quota_iunit_sz — referring to kilo bytes 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_*.

They can also be set as an option to lmc --quota. Changes will be required while using the lconf command with the parameter write_conf. A command like lconf --write_conf is to be used on the MDS. 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.

The values set for the MDS must match the values set on the OSTs.

The parameter quota_bunit_sz displays bytes, however lfs setquota uses kilo bytes. The parameter quota_bunit_sz must be a multiple of 1024. A proper minimum bkilo byte size for lfs setquota can be calculated by:

Size in bkilo bytes = (quota_bunit_sz * (number of OSTS + 1)) / 1024.

We add one to the number of OSTs as the MDS also consumes bkilo bytes. 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 the file creation.

CHAPTER II – 8. RAID

8.1 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 gives a survey of the issues and recommendations.

8.1.1 Reliability

Given below is a quick calculation that leads to the conclusion that without any further redundancy RAID5 is not acceptable for large clusters and RAID6 is a must.

Take a 1PB file system - that is 2000 disks of 500GB capacity. The MTF of a disk is likely about 1000 days and repair time at 10% of disk bandwidth is close to 1 day (500GB at 5MB/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 the 1 day of rebuilding the chance that a second disk in the same array fails is about 9 / 1000 ~= 1/100. This means that the in the expected period of 50 days a double failure in a RAID5 stripe will lead to data loss.

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

8.1.2 Selecting Storage for the MDS and OSS

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

Having disk monitoring software in place so that rebuilds happen without any delay should be regarded as mandatory. We recommend backups of the meta-data file systems. 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. These reduce RAID recovery time from a rebuild to a quick resynchronization.

8.1.3 Understanding Double Failures with Hardware and Software RAID5

Software RAID does not offer the hard consistency guarantees of top-end enterprise RAID arrays. Those guarantees state 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 is functioning without disk failures, but experiencing sudden power down events, such interrupted writes on journal file systems can affect file data and data in the journal. Meta data itself is re-written from the journal during recovery and will be 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, but this is a known problem with incomplete writes and caches anyway. Hence recovery of the disk file systems with software RAID is similar to recovery without software RAID. Moreover, using Lustre servers with disk file systems does not change these guarantees.

Problems can arise if after an abrupt shutdown a disk fails on restart. In this case even single block writes provide no guarantee that, for example, the journal will not be corrupted.

Hence:

1. IF A POWERDOWN IS FOLLOWED BY A DISK FAILURE, THE DISK FILE SYSTEM NEEDS A FILE SYSTEM CHECK.

2. IF A RAID ARRAY DOES NOT GUARANTEE before/after SEMANTICS, the same requirement holds.

We believe this requirement is present for most arrays that are used with Lustre, including the successful and popular DDN arrays.

CFS will release a modification to the disk file system 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.

8.1.4 Performance considerations

CFS is currently improving the Linux software RAID code to preserve large I/O which the disk subsystems can do very efficiently. With the 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.

8.1.5 Formatting

To format a software RAID file system, use the stride_size option while formatting.

8.2 Disk Performance Measurement

Below are some tips and insights for disk performance measurement. Some of this information is specific to RAID arrays and/or the Linux RAID implementation.

1. 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.

2. Verify drive ordering and identification.

For example, on a test system with a Marvell driver, the disk ordering is not preserved between boots but the controller ordering is. Therefore, we had to perform the sgp_dd survey and create arrays without rebooting.

3. Disks and arrays are very sensitive to request size.

To identify the most ideal request size for a given disk, benchmark the disk with different record sizes ranging from 4 KB to 1-2 MB.

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

To properly handle IO request sizes greater than 256 KB, the current Linux kernel either needs a driver patch or some changes in the block layer defaults, namely MAX_SECTORS, MAX_PHYS_SEGMENTS and MAX_HW_SEGMENTS. CFS kernels contain this patch. See blkdev_tunables-2.6-suse.patch in the CFS source.

5. I/O scheduler

Try different I/O schedulers because their behavior varies with storage and load. CFS recommends the deadline or noop schedulers. Benchmark them all and choose the best one for your setup. For further information on I/O schedulers, visit the following URLs:

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

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

6. 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
```

Both the above outputs were achieved on the same disk with the same parameters for sgp_dd. The only difference is that in the first case /dev/sda was used; while 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. The numbers do not change on testing with different I/O schedulers.

> **NOTE:** The sg device cannot be used by Lustre as it is not a block device – the sg device is used for performance measurement only.

7. Requests with partial-stripe write impair RAID5.

Remember that RAID 5 in many cases will do a read-modify-write cycle, which is not performant.

Try to avoid synchronized writes. Probably subsequent writes would make the stripe full and no reads will be needed. Try to configure RAID5 and the application in such a manner that most of the writes will be full-stripe and stripe-aligned.

8. NR_STRIPES in RAID5 (Linux kernel parameter)

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

9. Do not put an ext3 journal onto RAID5.

As journal is written linearly and synchronously, in most cases writes will not fill whole stripes. In this case, RAID5 will have to read parities.

10. Suggested MD device setups for maximum performance:

MDT

- RAID1 with internal journal and 2 disks from different controllers
- If you require larger MDTs, create 2 equal-sized RAID0 arrays from multiple disks. Create a RAID1 array from these 2 arrays. Using RAID10 directly requires a newer mdadm (the tool that administers software RAID on Linux) than the one shipped with RHEL 4. You can also use LVM instead of RAID0, but this has not been tested.

OST

- File system: RAID5 with 6 disks, each from a different controller.
- External journal: RAID1 with 2 partitions of 400MB (or more), each from disks on different controllers. FIXME

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

  To enable an external journal, you can use the above options in the lmc script used to create your XML. mdX is the external journal device.

  Before running --reformat, setup the journal device (/dev/mdX) by running:

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

- You can create a root file system, swap, and other system partitions on a RAID1

array with partitions on any 2 remaining disks. The remaining space on the OST journal disk could be used for this.

CFS has not tested RAID1 of swap.

11. rsz in sgp_dd:

It must be equal to the multiplication of <chunksize> and (disks-1).

You also should pass stripe=N, and extents or mballoc as a mountfs option for OSS. Here N = <chunksize> * (disks-1) / pagesize.

12. Run fsck on power failure or disk failure (RAID arrays).

- You must run fsck on an array in the event of a power failure and failure of a disk in the array due to potential write consistency issues.
- You can automate this in rc.sysinit by detecting degraded arrays.

8.2.1 Sample Graphs

8.2.1.1 Graphs for Write Performance:

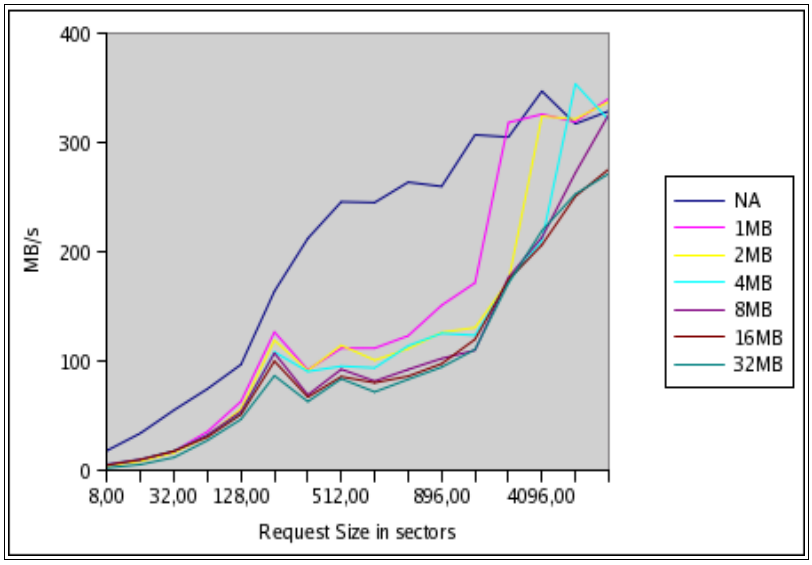

Figure 2.9.1: Write - RAID0, 64K chunks, 6 spindles

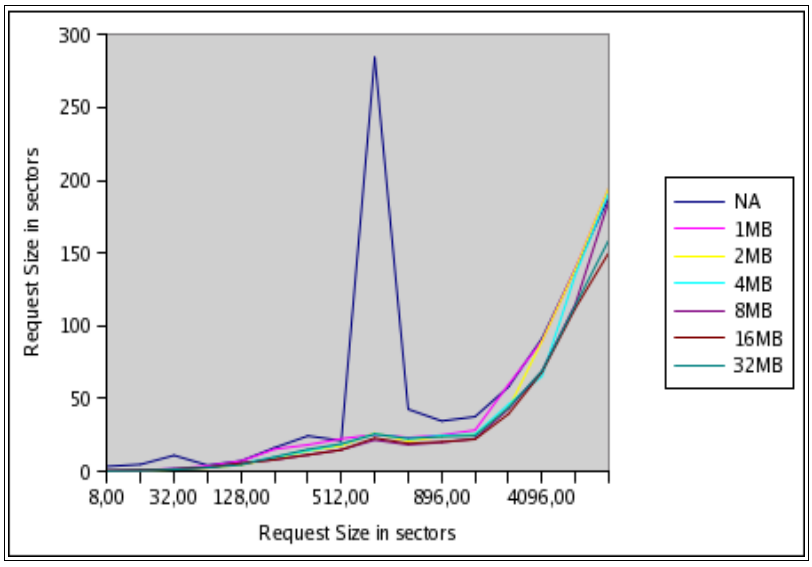

Figure 2.9.2: Write - RAID5, 64K chunks, 6 spindles

8.2.1.2 Graphs for Read Performance:

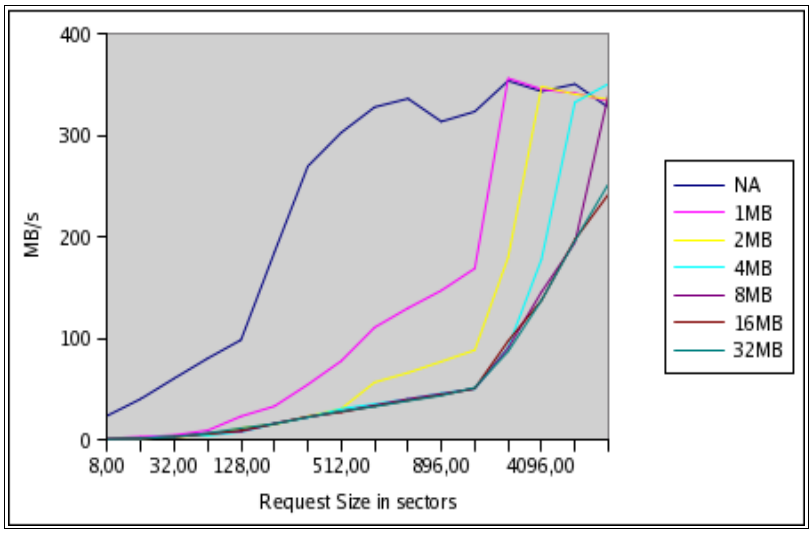

Figure 2.9.3: Read - RAID0, 64K chunks, 6 spindles

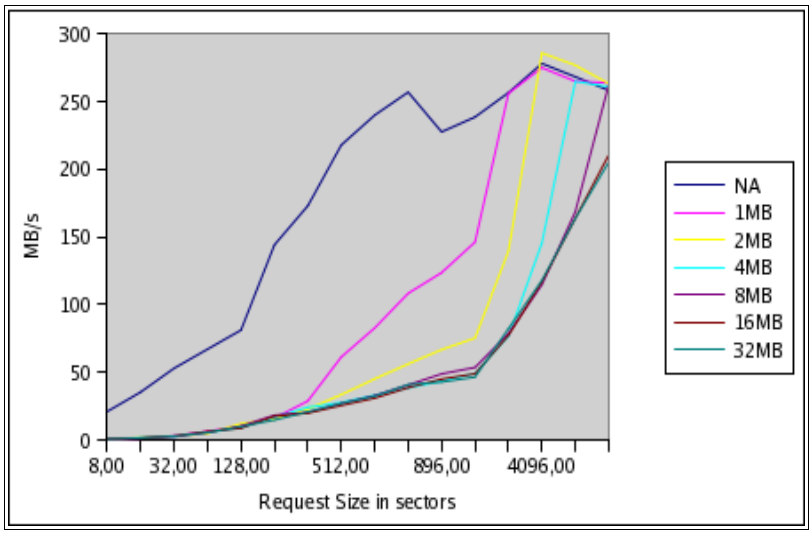

Figure 2.9.4: Read – RAID5, 64 K chunks, 6 spindle

CHAPTER II – 9. BONDING

9.1 Network Bonding

Bonding is a method of aggregating multiple physical links into a single logical link. This technology is also known as trunking, port trunking and link aggregation. We will use the term bonding.

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. Consult IEEE for more information.)

9.2 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 this support. The network driver for the interfaces to be bonded must have the ethtool support. The ethtool support is necessary for determination of the slave speed and duplex settings. All recent network drivers implement it.

To verify that your interface supports ethtool:

```
$ which ethtool

$ ethtool eth0

Settings for eth0:

Supported ports: [ MII ]

Supported link modes:   10baseT/Half 10baseT/Full \
100baseT/Half100baseT/Full1000baseT/Half1000baseT/Full

Supports auto-negotiation: Yes

(ethtool will return an error if your card is not supported.)
```

To quickly check whether your kernel supports bonding:

```
$ grep ifenslave /sbin/ifup

$ which ifenslave
```

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

9.3 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 setting 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 the failure of an interface transparent to avoid serious network degradation during link failures. 100 milliseconds is a reasonable default. Increase the timeout for a busy network.

```
$ miimon=100
```

9.4 Setup

Follow the process below to setup bonding:

Create a virtual 'bond' interface.

Assign an IP address to the 'bond' interface.

Attach one or more **slave** interfaces to the **bond** interface. Typically the MAC address of the first slave interface will become the MAC address of the bond.

Setup the bond interface and its options in /etc/modprobe.conf. Start the slave interfaces by 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.

Our examples are from Red Hat systems, and use /etc/sysconfig/networking-scripts/ifcfg-* for setup. The OSDL reference site given below includes detailed instructions for other configuration methods, instructions for using DHCP with bonding, and other setup details. We strongly recommend using this site.

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

Check /proc/net/bonding to determine status on bonding. There should be a file there for each bond interface. Check the interface state with ethtool or ifconfig. ifconfig lists the first bonded interface as “bond0.”

9.4.1 Examples

Let us see an example of Modprobe.conf for bonding ethernet interfaces eth1 and eth2 to bond0:

```
install bond0 /sbin/modprobe -a eth1 eth2 && /sbin/modprobe bonding \
miimon=100 mode=802.3ad xmit_hash_policy=layer3+4

alias bond0 bonding
```

ifcfg-bond0

```
DEVICE=bond0

BOOTPROTO=static

IPADDR=###.###.##.##
(Assign here the IP of the bonded interface.)

NETMASK=255.255.255.0

ONBOOT=yes
```

ifcfg-eth1 (eth2 is a duplicate)

```
DEVICE=eth1 # Change to match device

MASTER=bond0
```

```
SLAVE=yes

BOOTPROTO=none

ONBOOT=yes

TYPE=Ethernet
```

From linux-net.osdl.org:

```
For example, the content of /proc/net/bonding/bond0 after the driver\
is loaded with parameters of mode=0 and miimon=1000 is generally as \
follows:

   Ethernet Channel Bonding Driver: 2.6.1 (October 29, 2004)

        Bonding Mode: load balancing (round-robin)

        Currently Active Slave: eth0

        MII Status: up

        MII Polling Interval (ms): 1000

        Up Delay (ms): 0

        Down Delay (ms): 0


        Slave Interface: eth1

        MII Status: up

        Link Failure Count: 1


        Slave Interface: eth0

        MII Status: up

        Link Failure Count: 1
```

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

NOTE: All the 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

bond0    Link encap:Ethernet  Hwaddr 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


 eth0     Link encap:Ethernet  Hwaddr 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


 eth1     Link encap:Ethernet  Hwaddr 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
```

9.5 Lustre Configuration

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 necessary, specify “bond0” using the Lustre *networks* parameter:

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

9.6 References

Below are some references that we recommend -

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

This is the most exhaustive reference and is highly recommended. It includes explanations of more complicated setups, including the use of DHCP with bonding.

CHAPTER II – 10. UPGRADING LUSTRE FROM 1.4 TO 1.6

10.1 Upgrading from 1.4.6 and later to 1.6

10.1.1 Upgrade Requirements

You must remember following important points before upgrading Lustre.

- You must upgrade MDT before OSTs.
- Upgrade procedure will be:
  i. lconf failover shutdown
  ii. install new modules
  iii. run tunefs.lustre
  iv. mount startup.
- Upgrade can be done across a failover pair, in which case procedure will be:
  i. install new modules on backup server
  ii. lconf failover shutdown
  iii. run tunefs.lustre on new server
  iv. mount startup on new server
  v. install new modules on primary server.
- The file system name must be less than or equal to 8 characters (so that it fits on the disk label).
- When upgrading a version older than 1.4.6, the OST indexes will not be found. Hence, specify the index # to tunefs.lustre.

10.1.2 Supported Upgrade Paths

Entire File System or individual servers/clients

1. Servers can undergo a "rolling upgrade", where individual servers (or their failover partners) and clients are upgraded one at a time and restarted, so that the file system never goes down. However, this prevents the ability to change certain parameters.
2. The entire file system can be shutdown, and all servers and clients upgraded at once.
3. Any combination of the above two paths.

Interoperability between the nodes

1. Clients
   i. Old live clients can continue to communicate with old/new/mixed servers.
   ii. Old clients can start up using old/new/mixed servers.
   iii. New clients can start up using old/new/mixed servers (use old mount format for old MDT).
2. OSTs
   i. New clients/MDTs can continue to communicate with old OSTs.
   ii. New OSTs can only be started after the MGS has been started (typically this means "after the MDT has been upgraded.")
3. MDTs
   i. New clients can communicate with old MDTs.
   iii. New co-located MGS/MDTs can be started at any point.
   iv. New non-MGS MDTs can be started after the MGS has been started.

10.1.3 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 will be the same):

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

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

10.1.4 Upgrading a Lone File System

tunefs.lustre will find the old client log on an 1.4.x MDT that is being upgraded to 1.6. (If the name of the client log is not "client", use the lustre_up14.sh script as described in steps 2-4 below.)

1. Shutdown the MDT –

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

2. Install the new version of Lustre.
3. Run tunefs.lustre to upgrade the old configuration. There are two options here:
4. Rolling upgrade keeps a copy of the original configuration log, allowing immediate reintegration into a live file system, but preventing OSC parameter and failover NID changes. (The writeconf procedure can be performed later to eliminate these

restrictions. For details, see the section **2.2.3.2 Writeconf** in **Part II – Chapter 2. Lustre Installation**.)

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

i. --writeconf begins a new configuration log, allowing permanent modification of all parameters (Refer section **4.1.6 Changing Parameters** in **Part III – Chapter 4. Lustre Troubleshooting and Tips**), but requiring all other servers and clients to be stopped at this point, and no clients can be started until all OSTs are upgraded.

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

5. Start the upgraded MDT –

```
mdt1# mount -t lustre /dev/sda1 /mnt/test/mdt
```

6. OSTs for this FS can now be upgraded and started in a similar manner, except they need the address of the MGS. Note that very old installations may also need to specify the OST index (for instance, --index=5).

```
ost1# tunefs.lustre --ost --fsname=testfs --mgsnode=mdt1 /dev/sdb
```

10.1.5 Upgrading Multiple File Systems with a Shared MGS

The actual requirement is MGS first, then for any single file system the MDT must be upgraded and mounted, and then the OSTs for that file system. If the MGS is co-located with the MDT, then 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 file systems), then the old config logs must be transferred to the MGS manually.

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

```
mgsnode# mkfs.lustre --mgs /dev/sda1
```

2. Mount the MGS disk as type ldiskfs.

```
mgsnode# mount -t ldiskfs /dev/sda1 /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# sh lustre_up14.sh /dev/sdb testfs
debugfs 1.35 (28-Feb-2004)
/dev/sda1: catastrophic mode - not reading inode or group bitmaps
Copying log mds1 to testfs-MDT0000. Okay [y/n]?y
Copying log cfs21 to testfs-client. Okay [y/n]?y
Copying log client to testfs-client. Okay [y/n]?y
ls -l /tmp/logs
total 24
-rw-r--r--  1 root root 9408 Jun  9 15:20 testfs-client
```

```
-rw-r--r--  1 root root 9064 Jun  9 15:20 testfs-MDT0000

mdt1# scp /tmp/logs/* mgsnode:/mnt/mgs/CONFIGS/
```

4. Unmount the MGS ldiskfs mount.

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

5. Start the MGS.

```
mgsnode# mount -t lustre /dev/sda1 /mnt/mgs
```

6. Shutdown one of the old MDTs.

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

7. Upgrade the old MDT.

```
install new Lustre 1.6

mdt1# tunefs.lustre --mdt --nomgs --fsname=testfs \
--mgsnode=mgsnode@tcp0 /dev/sdb
```

(--nomgs is *required* for upgrading a non-co-located MDT.)

8. Start the upgraded MDT.

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

9. OSTs for this FS can now be upgraded and started.

```
ost1# lconf --failover --cleanup config.xml

install new Lustre 1.6

ost1# tunefs.lustre --ost --fsname=testfs \
--mgsnode=mgsnode@tcp0 /dev/sdc

ost1# mount -t lustre /dev/sdc /mnt/test/ost1
```

10. Other MDTs can be upgraded in a similar manner. Bear two things in mind:

    vii. The MGS must **NOT** be running (mounted) when the backing disk is mounted as ldiskfs.

    ii. The MGS **MUST** be running when first starting a newly-upgraded server (MDT or OST).

10.2 Downgrading to 1.4.6/7 from 1.6

10.2.1 Downgrade Requirements

- The file system must have been upgraded from 1.4.x. In other words, a file system created or reformatted under 1.6 cannot be downgraded.
- Any new OSTs that were dynamically added to the file system will be unknown under 1.4.x. Potentially it is possible to add them back using lconf --write-conf, but care must be taken to use the correct UUID of the new OSTs.
- Downgrading an MDS that is also acting as an MGS will prevent access to all other file systems that the MGS was serving.

10.2.2 Downgrading a File System

4. Shutdown all clients.
5. Shutdown all servers.
6. Install Lustre 1.4.x on the client and server nodes.
7. Restart the servers (OSTs, then MDT) and clients.

NOTE: All the OST additions and parameter changes made since the file system was upgraded will be lost.

PART III. LUSTRE TUNING, MONITORING AND TROUBLESHOOTING

CHAPTER III – 1. LUSTRE I/O KIT

1.1 Prerequisites

The Lustre I/O kit is a collection of benchmark tools for a Lustre cluster. You can download the I/O kits from:

https://downloads.clusterfs.com/customer/lustre-iokit/

In this directory, you will find two packages. The 'scali-lustre-iokit' is a Python tool maintained by the kind team at Scali, and is not discussed in this version of the manual. The 'lustre-iokit' package consists of a set of scripts developed and supported by CFS.

Prerequisites for the CFS I/O kit:

- password-free remote access to nodes in the system (Normally obtained via ssh or rsh)
- Lustre file system software
- sg3_utils for the sgp_dd utility

The kit can be used to validate the performance of the various hardware and software layers in the cluster and also as a way of finding and troubleshooting input/output issues.

It is very important to establish performance from the “bottom up” perspective. Firstly, the performance of a single raw device should be verified. Once this is completed, you should then 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, the other software layers can be added and tested in an incremental manner.

The kit contains three tests. The first surveys basic performance of the device and bypasses the kernel block device layers, buffer cache and file system. The subsequent tests survey progressively higher layers of the Lustre stack. Typically with these tests, Lustre should deliver 85-90% of the raw device performance.

1.2 Running the I/O Kit Tests

As mentioned above, the I/O kit bundle contains three testing tools:

- sgpdd survey
- obdfilter survey
- ost survey

1.2.1 sgpdd_survey

This is the tool for testing the **bare metal** performance, while bypassing as much of the kernel as we can. It does not require Lustre software, but does require the sgp_dd package. 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 input/output. 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 mentioned 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 this tool, note that raw device 0 can not 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.

The 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 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 contain lines like:

```
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 the 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 sgp_dd is unlikely to be able to allocate input/output buffers, "ENOMEM" is printed.

If all the sgp_dd instances do not successfully report a bandwidth number, "failed" is printed.

> **NOTE:** This test overwrites the device being tested and will result in the LOSS OF ALL DATA on that device. Exercise caution when selecting the device to be tested.

1.2.2 obdfilter_survey

This survey script processes sequential input/output 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 Lustre components below.

1. The 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 all the names of the obdfilter instances, which should already be up and running. If some are on different nodes, you also need to specify their host names, 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.

2. The 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 all the names of the echo_client instances, which should already be up and running. If some are on different nodes, you also need to specify their host names, for example, node1:ECHO_node1.

3. The 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 with above, you need to tell the script all the names of the 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.

Running the script

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

Running the 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, use:

```
$ lconf --refomat <XML file>
```

Remember, write tests are destructive. This test should be run prior to startup of your actual Lustre file system. 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 the clients. They 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 *client_names* undefined.

For example:

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

Running the script against a network

If you are driving obdfilter or obdecho instances over the network, you must instantiate the echo_clients yourself using lmc/lconf. Set the shell array variable *client_names* to the names of the echo_client instances and leave *ost_names* undefined.

You can optionally prefix any name in *ost_names* or *client_names* with the host name that it is running on, for example, *remote_node:ost4*. If you are running remote nodes, you need to ensure the following:

- custom_remote_shell() works on your cluster
- all pathnames you specify in the script are mounted on the node you start the survey from and on all the remote nodes

- obdfilter-survey must be installed on the clients at the same location as on the master node

1. First, 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 host names, or *hostname* of the current node will be 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:

```
$ lconf --node (hostname)/(path)/echo.xml
$ 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:

```
$ lconf --node client /(path)/echo.xml
$ 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:

```
$ lctl ping SERVER_NID
```

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

For example:

```
$ 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 done, cleanup echo_client/obdecho instances
   - on clients:

```
$ lconf --cleanup --node client /(path)/echo.xml
```

- on server(s):

```
$ lconf --cleanup --node (hostname)/(path)/echo.xml
```

7. 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 the 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.

Script output

The summary file and stdout contain lines like:

```
ost 8 sz 67108864K rsz 1024 obj    8 thr    8 write   613.54 \
[ 64.00, 82.00]
```

Where:

ost 8 is the total number of OSTs under test

sz 67108864K is the total amount of data read or written (in KB)

rsz 1024 is the record size (size of each echo_client input/output)

obj 8 is the total number of objects over all OSTs

thr 8 is the total number of threads over all OSTs and objects

write is the test name. If more tests have been specified they all appear on the same line

613.54 is the aggregate bandwidth over all OSTs measured by dividing the total number of MB by the elapsed time

[64.00, 82.00] are the minimum and maximum instantaneous bandwidths seen on any individual OST.

Note that 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.

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 input/output sizes during each test. These numbers help find pathologies in the system when the file system block allocator or the block device elevator fragment I/O requests.

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

1.2.3 ost_survey

This is a shell script that uses lfs setstripe to perform input/output against a single OST. It will write a file (currently using dd) to each OST in the Lustre file system, comparing read and write speeds. It is used to detect misbehaving disk subsystems. Note that we have frequently discovered wide performance variations across all LUNs in a cluster.

To run the script, supply a file size in KB and the Lustre mount point.

For example:

```
$ ./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
```

CHAPTER III – 2. LUSTREPROC

2.1 Introduction

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

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

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

2.1.1 /proc Entries for Lustre

2.1.1.1 Recovery

/proc/sys/lustre/upcall

This will contain the path of the recovery upcall or DEFAULT for the normal case where there is no upcall. Certain states will place information here, including

- FAILED_IMPORT – tgt_uuid obd_uuid net_uuid – which indicates failure of an upcall. The UUID information identifies target, obd name and network.
- RECOVERY_OVER tgt_uuid – the upcall called on the server when the recovery period has ended. The UUID is the target that was in recovery mode. For example, syslog message:

```
"May 25 13:35:46 d2_q_0 kernel: Lustre: \
12162:0:(recover.c:77:ptlrpc_run_recovery_over_upcall()) Invoked \
upcall DEFAULT RECOVERY_OVER ost-alpha_UUID"
```

/proc/sys/lnet/upcall

- LBUG src_file line_number function – which is called when an LBUG occurs.

The script paths can be configured with lmc and/or lconf or by modifying the corresponding *proc* entries. Setting an upcall to "DEFAULT" means that the recovery will be handled within the kernel by reconnecting to the same device.

2.1.1.2 Lustre Timeouts/ Debugging

/proc/sys/lustre/timeout

This is the time period for which a client will wait on a server to complete an RPC (default 100s). Servers will 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 1MB) to complete. The client will ping recoverable targets (MDS and OSTs) at one quarter of the timeout and the server will wait 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 (halft 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 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.

2.1.1.3 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
10.67.73.181@tcp        1    up     8     8     8     8     7   0
```

Fields are explained below:

refs – A reference count, used for debugging primarily

state – Up or down

max – Maximum number of concurrent sends from this peer

rtr – Routing buffer credits

min – Minimum routing buffer credits seen

tx – Send credits

min – Minimum send credits seen

queue – Total bytes in active/queued sends.

Credits work like a semaphore. At start they are initialized to allow a certain number of operations (8 in this example). LNET keeps a track of the minimum value so that you can see how congested a resource was.

If **rtr/tx** is less than **max**, there are operations in progress. The number of operations is equal to **rtr** or **tx** subtracted from **max**.

If **rtr/tx** is greater that **max**, there are operations blocking.

LNET also limits concurrent sends and router buffers allocated to a single peer so that no peer can occupy all these resources.

/proc/sys/lnet/nis

Shows current queue health on this node.

Fields are explained below:

nid – The network interface

refs – Internal reference counter

peer – the number of peer-to-peer send credits on this NID. Credits are used to size various 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.

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
```

2.2 Input/output

/proc/fs/lustre/llite/fs0/max_read_ahead_mb

This file contains the size of the client per-file read-ahead (default 40 MB). Setting this to zero will disable readahead.

/proc/fs/lustre/llite/fs0/max_cache_mb

This is the maximum amount of inactive data cached by the client (default 3/4 of RAM).

2.2.1 Client Input/output RPC Stream Tunables

The Lustre engine will always attempt to pack an optimal amount of data into each input/output RPC and will attempt to keep a consistent number of issued RPCs in progress at a time. Lustre exposes several tuning variables to adjust behaviour 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

blocksize           filesfree          max_dirty_mb  \
ost_server_uuid             stats
```

... and so on

The files related to tuning the RPC stream are as follows:

/proc/fs/lustre/osc/<*object name*>/max_dirty_mb

This controls how many megabytes 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 will 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 zero and 512 are allowed. If zero is given, no writes will be cached, but unless you use large writes (1MB or more) performance will suffer noticably.

/proc/fs/lustre/osc/<*object name*>/cur_dirty_bytes

This 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 value represents the maximum number of pages that will undergo input/output in a single RPC to the OST. The minimum is a single page and the maximum for this setting is platform depedent (256 for i386/x86_64, possibly less for ia64/PPC with larger PAGE_SIZE), though generally amounts to a total of one megabyte in the RPC.

/proc/fs/lustre/osc/<*object name*>/max_rpcs_in_flight

This value represents 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 one and maximum 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*> will vary depending on the specific Lustre configuration. See the sample output from the commands for examples of <*object name*>.

2.2.2 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/OSC_uml0_ost1_MNT_localhost/rpc_stats
snapshot_time:         1067551484:37103 (secs:usecs)
RPCs in flight:        0
pending write pages:   0
pending read pages:    0

other RPCs in flight when a new RPC is sent:
0:              0
1:              0
2:              0
3:              0
4:              0
5:              0
6:              0
7:              0
8:              0
9:              0
10:             0
```

```
11:              0
12:              0
13:              0
14:              0
15:              0

pages in each RPC:
0:               0
1:               0
2:               0
3:               0
4:               0
5:               0
6:               0
7:               0
8:               0
9:               0
10:              0
11:              0
12:              0
13:              0
14:              0
15:              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\input/output 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.

2.2.3 Watching the OST Block Input/output Stream

Similarly, there is a "brw_stats" histogram in the obdfilter directory which shows you the statistics for number of input/output requests sent to the disk, their size and whether they are contiguous on the disk or not.

```
cat /proc/fs/lustre/obdfilter/OST_localhost/brw_stats

snapshot_time:         1089922302:248138 (secs:usecs)


                     read                     write
pages per brw     brws  % cum % |    rpcs   %  cum  %
1:                   0   0   0   |       1   0   0
2:                   0   0   0   |       0   0   0
4:                   0   0   0   |       0   0   0
8:                   0   0   0   |       0   0   0
16:                  0   0   0   |       0   0   0
32:                  0   0   0   |       0   0   0
64:                  0   0   0   |       0   0   0
128:                 0   0   0   |     140  99 100


                     read                     write
discont pages     rpcs  % cum % |    rpcs   %  cum  %
0:                   0   0   0   |     141  100 100


                     read                     write
discont blocks    rpcs  % cum % |    rpcs   %  cum  %
0:                   0   0   0   |     123  87  87
1:                   0   0   0   |      18  12 100
```

pages per brw = number of pages per RPC request, which should match aggregate client rpc_stats

discont pages = number of discontinuities in the logical file offset of each page in a single RPC

discont blocks = number of discontinuities in the physical block allocation in the file system for a single RPC

2.2.4 mballoc History

/proc/fs/ldiskfs/loop0/mb_history

Each mballoc-enabled partition will have this file.

Sample output:

```
pid inode goal result found grps cr merge tail broken
1593 25052 1/12289/255 1/12289/255 1 0 0 M 0
0
1591 25052 1/12544/256 1/12544/256 1 0 0 M 0
0
1592 25052 1/12800/256 1/12800/256 1 0 0 M
256 512
1590 25052 1/13056/256 1/13056/256 1 0 0 M 0
0
1593 25052 1/13312/256 1/13312/256 1 0 0 M
256 1024
1591 25052 1/13568/256 1/13568/256 1 0 0 M 0
0
1592 25052 1/13824/256 1/13824/256 1 0 0 M
256 512
1590 25052 1/14080/256 1/14080/256 1 0 0 M 0
0
1593 25052 1/14336/256 1/14336/256 1 0 0 M
256 2048
1592 25052 1/14592/256 1/14592/256 1 0 0 M 0
```

Fields:

pid = Process that made the allocation

inode = inode number allocated blocks

goal = inital request that came to mballoc (group/block-in-group/number-of-blocks)

result = what mballoc actually found for the 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** – the 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)

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 zero or one 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 few dozens often either your disk file system is pretty fragmented or mballoc is doing something wrong in the group selection part.

2.3 Locking

/proc/fs/lustre/ldlm/ldlm/namespaces/<OSC name|MDC name>/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 neighbourhood 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.

2.4 Debug Support

/proc/sys/lnet/debug

Setting this to zero will completely turn-off debug logs for all the debug types. While setting it to -1 will turn on full debugging (see D_* definitions in lnet/include/linux/libcfs.h).

/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:** Above entries exist only 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>
```

2.4.1 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/llite/fs0/stats
/proc/fs/lustre/mdt/MDT/mds_readpage/stats
/proc/fs/lustre/mdt/MDT/mds_setattr/stats
/proc/fs/lustre/mdt/MDT/mds/stats
/proc/fs/lustre/osc/OSC_uml0_ost3_MNT_localhost/stats
/proc/fs/lustre/osc/OSC_uml0_ost2_MNT_localhost/stats
/proc/fs/lustre/osc/OSC_uml0_ost1_MNT_localhost/stats
/proc/fs/lustre/osc/OSC_uml0_ost3_mds1/stats
/proc/fs/lustre/osc/OSC_uml0_ost2_mds1/stats
/proc/fs/lustre/osc/OSC_uml0_ost1_mds1/stats
/proc/fs/lustre/obdfilter/ost2/stats
```

```
/proc/fs/lustre/obdfilter/ost3/stats

/proc/fs/lustre/obdfilter/ost1/stats

/proc/fs/lustre/ost/OSS/ost_create/stats

/proc/fs/lustre/ost/OSS/ost/stats

/proc/fs/lustre/ldlm/ldlm/ldlm_canceld/stats

/proc/fs/lustre/ldlm/ldlm/ldlm_cbd/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.pl /proc/fs/lustre/ost/OSS/ost/stats 10

/proc/fs/lustre/ost/OSS/ost/stats @ 1126198063.790389

Name            Cur.Count  Cur.Rate   #Events     Unit         last \
min       avg     max     stddev

req_waittime       12         0        1522       [usec]   19800.50 \
68   1135.52  242393   10297.09

req_qdepth         12         0        1522       [reqs]       0.58 \
0      0.15       3      0.45

req_active         12         0        1522       [reqs]       1.08 \
1      1.01       2       0.09

reqbuf_avail       12         0        1522       [bufs]      63.67 \
63     63.93      64       0.26

ost_setattr         0         0           2       [usec]       0.00 \
240    257.50     275      24.75

ost_read            0         0         220       [usec]       0.00 \
530   1262.77   74463    4972.71

ost_write           0         0         230       [usec]       0.00 \
1438   2200.02   28189    2342.42

ost_create          2         0          24       [usec]     274.00 \
72   7322.46   35521   12654.60

ost_destroy       400        18        1047       [usec]     736.09 \
626   1134.41   30260    1560.68

ost_get_info        0         0           2       [usec]       0.00 \
71     101.50     132      43.13

ost_connect         2         0          26       [usec]    1395.50 \
1170   5037.04   27153    7231.62

ost_set_info        2         0          24       [usec]     297.50 \
108    300.38    1162     208.49

ldlm_enqueue        0         0         474       [usec]       0.00 \
194    351.57    1911     154.21

obd_ping            4         0         294       [usec]     151.50 \
62    175.97     600      49.36
```

Where:

Cur.Count = the number of events of each type sent in the last interval (10s in this case)

Cur.Rate = the number of events per second in the last interval

#Events = the total number of such events since the system was started

Unit = the unit of measurement for that statistic (microseconds, requests, buffers)

last = the average rate of these events (in units/event) for the last interval during which they arrived. For instance, in the above mentioned case of ost_destroy it took an average of 736 microseconds per destroy for the 400 object destroys in the previous 10s

min = the minimum rate (in units/event) since the service started

avg = the average rate

max = the maximum rate

stddev = the standard deviation (not measured in all the cases)

The events common to all services are:

req_waittime — the amount of time a request waited in the queue before being handled by an available server thread

req_qdepth — the number of requests waiting to be handled in the queue for this service

req_active — the number of requests currently being handled

reqbuf_avail — the number of unsolicited lnet request buffers for this service

Some service specific events of interest are:

ldlm_enqueue — the time it takes to enqueue a lock (this includes file open on the MDS)

mds_reint — the time it takes to process an MDS modification record (includes create, mkdir, unlink, rename, setattr)

CHAPTER III – 3. LUSTRE TUNING

3.1 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").

3.1.1 OST Threads

The *ost_num_threads* option allows the number of OST service threads to be specified at module load time on the OSS nodes:

options ost ost_num_threads={N}

Number of OST threads is a function of the server capacity (RAM + CPUs). For a 2GB 2-CPU system this works out to be 64 OST service threads. For larger servers this might be as high as 512 threads. Giving a specific thread count via the module parameter *ost_num_threads*= overrides the default calculation.

Increasing the size of the thread pool may help when:

- several OSTs are exported from a single OSS
- the back-end storage is running synchronously
- input/output completions are taking excessive time.

In such cases, a larger number of input/output threads allows the kernel and storage to aggregate many writes together for more efficient disk input/output. The OST thread pool is shared — each thread allocates approximately 1.5 MB (maximum RPC size + 0.5 MB) for internal input/output buffers.

However, do note that memory consumption should be considered when increasing the thread pool size.

3.1.2 MDS Threads

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

options mds mds_num_threads={N}

At this time, no testing has been done as to what the optimal number of MDS threads are. The default number varies based on the server size up to a maximum of 32. The maximum number of threads (MDS_MAX_THREADS) is 512.

NOTE: The OSS and MDS will 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 this section and the above section **3.1.1 OST Threads**).
Setting the _mu_threads module parameter will disable the automatic thread creation behavior.

3.1.3 LNET Tunables

Transmit and receive buffer size:

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 (0) the system will automatically tune the transmit and receive buffer size. In almost every case, the defaults will produce the best performance. Do not attempt to tune this unless you are a network expert!

irq_affinity

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

If you have an SMP platform with a single fast interface such as 10GB Ethernet and more than two CPUs, you may see performance improve by turning this parameter off, as always test to compare the impact.

3.2 DDN Tuning

This section provides a guideline to configure DDN storage arrays for use with Lustre.

3.2.1 Settings

3.2.1.1 Segment Size

The cache segment size noticeably affects input/output performance. You should set the cache segment size differently on the MDT (which does small, random input/output) and on the OST (which does large, contiguous input/output). The optimum values we have found in customer testing are 64KB for the MDT and 1MB for the OST.

The necessary DDN client commands are given below.

For MDT LUN:

```
$ cache size=64

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

For OST LUN:

```
$ cache size=1024
```

3.2.1.2 maxcmds

In a particular case, changing this value from the default two to four has improved the write performance by as much as 30%. This works only with SATA-based disks and when only one controller of the pair is actually accessing the shared LUNs.

However, this recommendation comes with a warning. DDN support do not recommend changing this setting from the default. By increasing the value to five, the same set up experienced some serious problems.

The necessary DDN client command is given below, where the default value is two.

```
$ disk   maxcmds=3
```

3.2.1.3 Write-back Cache

Some customers run with the write-back cache turned on, because it significantly improves performance. They are willing to take the risk that when there is a DDN controller crash and they need to run e2fsck, it will take them less time than the performance hit from running with the write-back cache turned off.

Other customers run with the write-back cache off for increased data security. However, some of these customers experience performance problems with the small writes during journal flush. In this mode it is highly beneficial to also increase the number of OST service threads "ost_num_threads=512" in /etc/modprobe.conf, if the OST has enough

RAM (about 1.5MB/thread is preallocated for I/O buffers). More input/output threads allow more input/output requests to be in flight waiting for the disk to complete the synchronous write.

This is a decision that you need to make yourself — there is a trade off between improved performance and running the slight risk of data loss and downtime in the 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.

3.2.1.4 Further Tuning Tips

Some tips we have drawn from testing at a large installation include:

- Use the full device instead of a partition (sda vs sda1). When using the full device, Lustre will write nice aligned 1MB chunks to disk. Partitioning the disk can destroy this alignment and will noticeably impact performance.
- Separate the EXT3 OST into 2 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)

```
$ {LMC} --add mds --node io1 --mds iap-mds –dev /dev/sdi \
--mkfsoptions "-j -J device=/dev/sdj" --failover --group iap-mds
```

Very important: We have proved that we need to create one OST per TIER especially in write through (see the illustration below). This is of concern if you have 16 tiers. You should create 16 OSTs consisting of one tier each instead of eight made of two tiers each.

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

For example — one OST per tier

```
LUN  Label  Owner  Status      Capacity  Block   Tiers   Tier  list
                               (Mbytes)  Size

-------------------------------------------------------------------

0              1   Ready         102400     512    1        1
1              1   Ready         102400     512    1        2
2              1   Ready         102400     512    1        3
3              1   Ready         102400     512    1        4
4              2   Ready [GHS]   102400    4096    1        5
5              2   Ready [GHS]   102400    4096    1        6
6              2   Critical      102400     512    1        7
7              2   Critical      102400    4096    1        8
10             1  Cache Locked       64     512    1        1
11             1  Cache Locked       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   Critical         64     512    1         7
17              2   Critical         64     512    1         8
```

System verify extent: 16MB

System verify delay: 30

3.3 Large-Scale Tuning for Cray XT and Equivalents

This information applies to the Cray XT3 Catamount nodes only. Ignore this section if you find it irrelevant. This section expects you to be familiar with the operation of this type of system. The following section explains the parameters used with the kptllnd module.

3.3.1 Network Tunables

Given the large number of clients and servers possible on these systems, tuning various request pools becomes quite important. CFS is in the process of making changes to the ptllnd module.

One tunable parameter for enhancing the performance is **max_nodes**. It is the maximum number of queue pairs, and therefore the maximum number of peers that the instance of the LND may communicate with. Set **max_nodes** at a value higher than the product of total number of nodes and maximum processes per node.

Max nodes > (Total # Nodes) * (max_procs_per_node)

If you set **max_nodes** less than the above mentioned product, Lustre throws an error. If you set it too high, excess memory will be consumed.

There are a few other tunables in the code that may impact performance.

max_procs_per_node – It is the maximum number of cores (CPUs) on a single Catamount node. Portals must know this value in order to properly clean up various queues. LNET is not notified directly when a catamount process aborts. The first news it gets of this is when a new catamount process with the same cray portals NID starts up and sends a connection request. If the number of processes with that cray portals NID would now exceed **max_procs_per_node**, LNET removes the oldest one to make space for the new one.

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 quite important.

Two other parameters for ptllnd are **ntx** and **credits**.

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 will be bounded by the number of servers. A console error message will appear if the **tx** pool is exhausted.

Credits determine how many sends are in-flight at once on ptllnd. The optimum is 8 requests in-flight per server. The default is 128, which should be proper for most applications.

CHAPTER III – 4. LUSTRE TROUBLESHOOTING AND TIPS

4.1 Tips

4.1.1 Setting SCSI IO Sizes

Some SCSI drivers default to a maximum IO size that is too small for good Lustre performance. CFS has fixed quite a few drivers, but you may still find some drivers giving unsatisfactory performance with Lustre. As the default 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 IO performance, and analyzing Lustre stats indicates that the IO is not 1 MB, check **/sys/block/<device>/queue/max_sectors_kb**. If it is less than 1024, set it to 1024 in order to improve the performance. If changing this value does not change the IO size as reported by Lustre, you may want to examine the SCSI driver code.

4.1.2 Write Performance Better Than Read Performance

The performance of write operations on a Lustre cluster is typically better than that of reads. When doing writes, all clients are sending write RPCs asynchronously. The RPCs are allocated and written to disk in the order of their arrival. This allows the back-end storage to aggregate the writes efficiently to disk in many cases.

In the read case, the reads from the clients may come in a different order and may need a lot of seeking to get read from the disk. This hampers the read throughput noticeably.

There is currently 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 (1MB) readahead for 1000 clients would consume 1GB of RAM).

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

4.1.3 OST Object Missing or Damaged

You will be shown the message “OST object missing or damaged (OST "ost1", object 98148, error -2)” when the object storage server fails to find an object, or finds a damaged object.

If the reported error is -2 (-ENOENT, or "No such file or directory"), then the object is missing. This could occur either because the MDS and OST are out of sync, or because an OST object was corrupted and deleted.

If you have recovered the file system from a disk failure by using e2fsck, unrecoverable objects may have been deleted or moved to /lost+found on the raw OST partition. Because files on the meta data server (MDS) still reference these objects, attempts to access them will 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 will produce 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 CFS, and we will investigate.

If the reported error is anything else (such as -5, "I/O error"), it likely indicates a storage failure. The low-level file system will return 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. Most likely, however, this object is lost forever, and 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, you should inspect this server for storage problems immediately.

4.1.4 OSTs Become Read-Only

If the SCSI devices are inaccessible to Lustre at the block device level, ext3 will remount the device read-only to prevent file system corruption. This is a normal behavior. The status in /proc/fs/lustre/healthcheck also shows "not healthy" on the affected nodes.

You must restart the Lustre services using these file systems in order to recover them from this problem. There is no other way to know the IO made to disk, and the state of the cache may be inconsistent with what is on disk.

4.1.5 Identifying Missing OST

If an OST is missing for any reason, you may need to know what files are affected.

But for this, the file system should still be operational, even though one OST is missing. So that from any mounted client node it will be 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 that clients and the MDS do not time out trying to contact it. On MDS and client nodes, execute:

```
# lctl dl
```

This will generate a list of devices, and find the OST device number.

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

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

If the OST later becomes available it needs to be reactivated by executing:

```
# lctl --device N activate
```

Determine all the files striped over the missing OST:

```
# lfs find -R -o {OST_UUID} /mountpoint
```

This will return a simple list of filenames from the affected file system.

You can read the valid parts of a striped file if necessary:

```
# dd if=filename of=new_filename bs=4k conv=sync,noerror
```

Otherwise, you can also delete these files with "unlink" or "munlink".

If you need to know specifically which parts of the file are missing data you first need to determine the striping pattern, which will include the index of the missing OST:

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

The following computation is used to determine which offsets in the file are

affected:

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

where:

C = stripe count,

S = stripe size,

X = index of bad ost for this file

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, ...}

If the file system cannot be mounted, currently there is no way that would parse meta data directly from an MDS. If the bad OST is definitely not starting, options for mounting the file system anyway are to provide a loop device OST in its place, or to replace it with a newly formatted OST. In that case the missing objects are created and will read as zero-filled.

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

4.1.6 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 file system*, you can simply add the parameters as a "--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 params and just use the new params specified, 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
```

4.1.7 Adding a Failover

Adding a failover server node to a live Lustre file system

```
$ lctl conf_param testfs-OST0000.failover.node=3@elan,\
192.168.0.3@tcp0
```

On other system you can verify

```
$ cat /proc/fs/lustre/osc/testfs-OST0000-osc/ost_conn_uuid
```

Servers and clients will immediately be able to use the failover node. Note that tcp addresses must be in dotted-quad form, not hostname form. Multiple failover hosts can be specified by repeating the failnode= parameter.

```
$ failover.mode=<"failout","failover">
```

Failout returns errors immediately; failover waits for recovery. Failover is the default.

4.1.8 Default Striping

lov.stripesize=<bytes>

lov.stripecount=<count>

lov.stripeoffset=<offset>

Change the default striping information –

On MGS

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

On MDT and clients

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

4.1.9 Erasing a File System

If you want to erase a file system, you should just run the following command on your targets –

```
$ "mkfs.lustre –reformat"
```

If you are using a separate MGS and want to keep other file systems defined on

that MGS, then you must set the "writeconf" flag (the name is historical) on the MDT for that file system. The "writeconf" flag causes the config logs to be erased - they will be regenerated the next time servers start.

Follow the steps below for setting the “writeconf” flag on the MDT:

8. Unmount all clients/servers using this file system

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

9. Erase the file system, presumably replace it with another file system

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

10. If you have a separate MGS (that you do not want to reformat), then add the "writeconf" flag to mkfs.lustre on the MDT:

```
$ mkfs.lustre --reformat --writeconf –fsname spfs --mdt \
--mgs /dev/sda
```

NOTE: If you have combined MGS/MDT, reformatting the MDT will reformat the MGS as well, and so all configuration information will be lost and can start building your new FS. Nothing needs to be done with old disks that would not be part of the new file system; just do not mount them.

Part IV. Lustre for Users

CHAPTER IV – 1. FREE SPACE AND QUOTAS

1.1 Querying File System Space

The command **lfs df** is used to determine the disk space available on a file system. It displays the amount of available disk space on the mounted Lustre file system and shows space consumption per-OST. If multiple Lustre file systems are mounted, a PATH may be specified, but is not required.

Options	Description
-h	--human-readable print sizes in human readable format (For instance: 1K, 234M, 5G)
-i, --inodes	Lists inodes instead of block usage

Examples

```
fc3:~$ lfs df

UUID               1K-blocks      Used   Available   Use%  \
Mounted on

mds-p_UUID           4399856    528200     3871656     12 \
/mnt/lustre[MDT:0]

ost-a_UUID         153834852  55804744    98030108     36 \
/mnt/lustre[OST:0]

ost-b_UUID         153834852  55927804    97907048     36 \
/mnt/lustre[OST:1]


filesystem summary: 307669704 111732548  195937156     36 \
/mnt/lustre

fc3:~$ lfs df -h

UUID                1K-blocks      Used   Available   Use% \
Mounted on

mds-p_UUID              4.2M     515.8K      3.7M      12 \
/mnt/lustre[MDT:0]

ost-a_UUID            146.7M      53.2M     93.5M      36 \
/mnt/lustre[OST:0]

ost-b_UUID            146.7M      53.3M     93.4M      36 \
/mnt/lustre[OST:1]


filesystem summary:   293.4M     106.6M    186.9M      36 \
/mnt/lustre

fc3:~$ lfs df -i

UUID                  Inodes     IUsed     Ifree   IUse%   \
Mounted on
```

```
mds-p_UUID           1257360    272869     984491    21 \
/mnt/lustre[MDT:0]
ost-a_UUID           19546112   257430    19288682       1 \
/mnt/lustre[OST:0]
ost-b_UUID           19546112   257430    19288682       1 \
/mnt/lustre[OST:1]


filesystem summary:   1257360   272869    984491       21 \
/mnt/lustre
```

1.2 Using Quota

The **lfs quota** command displays disk usage and quotas. Only user quotas are displayed by default or with the **-u** flag.

A root user may use the **-u** flag with the optional *user* parameter to view the limits of other users. Users without the root user authority can view the limits of groups (of which they are members) by using the **-g** flag with the optional *group* parameter.

> **NOTE:** If a particular user has no files in a file system on which they have a quota, the command will show *quota: none* for that user. The user's actual quota is displayed when the user has files in the file system.

Examples

To display your quotas as a user “bob,” enter:

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

The above example will display the disk usage and limits for the user "bob."

To display quotas as the root user for user “bob,” enter:

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

The system can also show the below information about the 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
```

CHAPTER IV – 2. STRIPING AND OTHER I/O OPTIONS

2.1 File Striping

Lustre stores files of one or more objects on object storage targets (OSTs). When a file is comprised of more than one object, Lustre will stripe the file data across them in a round-robin fashion. The number of stripes, the size of each stripe and the servers chosen are all configurable.

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.

2.1.1 Advantages of Striping

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

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 such as these you want to 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 that much OSS bandwidth. The largest useful stripe count is bounded by the input/output rate of your clients/jobs divided by the performance per OSS.

The second reason to stripe is when a single object storage target (OST) does not have enough free space to hold the entire file.

2.1.2 Disadvantages of Striping

There are two disadvantages to striping which should deter you from choosing a default policy which stripes over all OSTs unless you really need it: increased overhead and increased risk.

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 can be performed in parallel, there is a big difference between doing one network operation and doing one hundred.

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 input/output. If each file has 100 objects, then the clients will all

compete with each other for the attention of the servers and the disks on each node will be seeking in 100 different directions. In this case, there is needless contention.

Increased risk is evident when you consider again 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 will be lost. By comparison, if every file has exactly one stripe, you will lose fewer files, but you will lose them in their entirety. Most users would rather lose some of their files entirely than all of their files partially.

2.1.3 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 tools enforce a multiple of 64 KB (the maximum page size on ia64 and PPC64 noeds), so that users on platforms with smaller pages do not accidentally create files which might cause problems for ia64 clients.

Although you could create files with a stripe size of 64 KB, this would be a poor choice. Practically, the smallest recommended stripe size is 512 KB because Lustre tries to batch input/output into 512 KB chunks over the network. This is a good amount of data to transfer at once. Choosing a smaller stripe size may hinder the batching.

Generally, a good stripe size for sequential input/output using high-speed networks is between 1 MB and 4 MB. Stripe sizes larger than 4 MB will 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 the stripe size with that in mind. If the file is written in a very consistent and aligned way, you can do it a favor by making the stripe size a multiple of the write() size.

The choice of stripe size has no effect on a single-stripe file.

2.2 Displaying Striping Information with lfs getstripe

Individual files and directories can be examined with **lfs getstripe**:

```
lfs getstripe <filename>
```

lfs will print the index and UUID for each OST in the file system along with the OST index and object ID for each stripe in the file. For directories, the default settings for files created in that directory will be printed.

A whole tree of files can also be inspected with **lfs** find:

```
lfs find [--recursive | -r] <file or directory> ...
```

2.3 lfs setstripe – Setting Striping Patterns

New files with a specific stripe configuration can be created with **lfs setstripe**:

```
lfs setstripe <filename> <stripe-size> <starting-ost> <stripe-count>
```

If you pass a stripe-size of **0**, the file system default stripe size will be used. Otherwise, the stripe-size must be a multiple of 16 KB.

If you pass a starting-ost of **-1**, a random first OST will be chosen. Otherwise the file will start on the specified OST index (starting at zero).

If you pass a stripe-count of **0**, the file system default number of OSTs will be 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 will be written to OST #0, until space is exhausted. This is probably not your intention. If you wish to adjust stripe-count only and keep the other parameters at their default, use this syntax:
> lfs setstripe 0 -1 <stripe_count>

2.3.1 Changing Striping for a Subdirectory

lfs setstripe works on directories to set a default striping configuration for files created within that directory. The usage is the same as for 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 the striping) Lustre will use those striping parameters instead of the file system default for the new file.

To change the striping pattern for a subdirectory, create a directory with desired striping pattern as described above. The subdirectories inherit the striping pattern of the parent directory.

> **NOTE**: Striping on directories only affects NEW files and NEW subdirectories created therein.

2.3.2 Using a Specific Striping Pattern for a Single File

lfs setstripe will create a file with a given stripe pattern.

lfs setstripe will fail if the file already exists.

2.4 Performing Direct Input/output

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 4k). If the alignment is not correct the call will return -EINVAL. Direct Input/output may help performance in cases where the client is doing a large amount of Input/output and is CPU-bound (CPU utilization 100%).

2.4.1 Making File System Objects Immutable

An immutable file or directory is one that cannot be modified, renamed or removed. To do this:

```
chattr +i <file>
```

chattr –i removes the flag

2.5 Other Input/output Options

2.5.1 MDS Space Utilization

Lustre comprises of large inodes, where each inode is at least 512 bytes by default. Lustre also needs sufficient space left for other metadata like journals (up to 400MB), bitmaps and directories. There are also a few regular files that Lustre uses to maintain cluster consistency. To be on the safer side we recommend you plan for 4KB per inode on the MDS.

If you use the **-i** option for mke2fs and if you are specifying some absolute number of inodes using **-N {num inodes}**, newer e2fsprogs will reduce the group size. This will allow an increased number of inodes beyond one inode per 1024 bytes. Every time you create a file on a Lustre file system, you might notice that one inode on the corresponding MDS (as well as one inode on the OST itself) is used. The minimum bytes per inode for ext3 are 1024 and the maximum block size is 4096. Thus the maximum ratio of inodes per block is four.

The file system on an MDS and that on an OST are independent of each other. Hence, the formatting parameters for the two need not be same. The size of the MDS file system solely depends on how many inodes you want in the total Lustre file system. It is not the size of the aggregate OST space. You can have a much higher maximum number of bytes per inode in the file system up to 128MB per eight inodes. This is useful for OSTs if you have a very large average file size.

As a result, the only important factor when calculating the MDS size is the average size of files to be stored in the file system. If the average file size is, for instance, 5MB and you have 100TB of usable OST space then you need at least (100 * 1024 * 1024 / 5) = 20 million inodes (though it is recommended to have twice the minimum, that is 50 million inodes). That means 4KB per inode space is the default. This works out to only 80GB of space for the MDS.

On the other hand, if you had a very small average file size, for example 4KB, iLustre is not very efficient. This is because you consume as much space on the MDS as you are consuming on the OSTs. This is not a very common configuration for Lustre. With a 2TB MDS you could potentially have 1KB per inode. It is not possible to have an inode of less than 512 bytes. So 2B inodes would need 2B * 4KB = 8TB of usable OST space. Depending on your needs, you could instead just do this with a single ext3 file system instead of Lustre.

> **NOTE:** In the Lustre file system, inodes are consumed and not the space.

2.5.2 End to End Client Checksums

To guard against data corruption, a Lustre client can perform end to end data checksums. This must be enabled on the individual client nodes. If the checksum is bad, the client will not have an IO error. The bad checksum will be reported immediately as a syslog message. Both client and OST will log messages at intervals showing that checksums are being validated. A /proc file controls the checksum behavior. The file is:

```
/proc/fs/lustre/llite/fs0/checksum_pages
```

To enable checksums on a client:

```
echo 1 > /proc/fs/lustre/llite/fs0/checksum_pages
```

CHAPTER IV – 3. LUSTRE SECURITY

3.1 Using Access Control Lists

An ACL, or access control list, is a set of data that informs an operating system about the permissions, or access rights, that each user or group has to a specific system object, such as a directory or file. Each object has a unique security attribute that identifies which users have access to it. The ACL is a list of each object and user access privileges such as read, write or execute.

3.1.1 How do ACLs work?

Implementing ACLs varies between operating systems. Systems that support the POSIX (Portable Operating System Interface) family of standards share a simple yet powerful file system permission model, which should be well-known to the Linux/Unix administrator. ACLs add finer-grained permissions to this model, allowing for more complicated permission schemes. For a detailed explanation of ACLs on Linux, we recommend the SuSE Labs article, “Posix Access Control Lists on Linux” found on-line here:

http://www.suse.de/~agruen/acl/linux-acls/online/

CFS has 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.

3.1.2 Lustre ACLs

Lustre versions 1.4.6 and above support POSIX ACLs. When using a Lustre client of version 1.4.5 or below with an MDS of version 1.4.6, or vice versa, the user space program generates an error “Operation not supported” during ACL operations.

The MDS needs to be configured in order to enable ACLs. This can be enabled when creating your configuration with *--mountfsoptions*:

```
 $ mkfs.lustre --fsname spfs --mountfsoptions=acl --mdt –mgs \
/dev/sda
```

Or, you can enable 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, this message will appear 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, any attempts to reference an ACL on a client will return an “Operation not supported” error.

3.1.3 Examples

These examples are taken directly from the POSIX paper referenced above. ACLs on a Lustre file system work exactly like ACLs on any Linux file system. They are manipulated with the standard tools in the standard manner. Here we create a directory and allow a specific user access.

```
[root@client spfs]# umask 027
[root@client spfs]# mkdir rain
[root@client spfs]# ls -ld rain
drwxr-x---  2 root root 4096 Feb 20 06:50 rain
[root@client spfs]# getfacl rain
# file: rain
# owner: root
# group: root
user::rwx
group::r-x
other::---

[root@client spfs]# setfacl -m user:chirag:rwx rain
[root@client spfs]# ls -ld rain
drwxrwx---+ 2 root root 4096 Feb 20 06:50 rain
[root@client spfs]# getfacl --omit-heade rain
user::rwx
user:chirag:rwx
group::r-x
mask::rwx
other::---
```

CHAPTER IV – 4. OTHER LUSTRE OPERATING TIPS

4.1 Expanding the File System by Adding OSTs

Instructions for adding OSTs to existing Lustre file systems

Step 1: Add a new ost by passing on the following commands

```
$ mkfs.lustre --fsname=spfs --ost --mgsnode=mds16@tcp0 /dev/sda
$ mkdir -p /mnt/test/ost0
$ mount -t spfs /dev/sda /mnt/test/ost0
```

Step 2: Possibly, migrate the data.

The file system will be quite unbalanced when new empty OSTs are added. New file creations will be automatically balanced. If this is a scratch file system or files are pruned at a regular interval 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 **mv** the temp file over the old one. Naturally, this should not be attempted with files which are currently being written to by users or applications. This operation will redistribute the stripes over the entire set of OSTs. A sample script for this migration is attached.

A very clever migration script would:

- 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 an enterprising Lustre administrator wants to explore this approach further, per-OST disk-usage statistics can be found under /proc/fs/lustre/osc/*.

Example Script:

```
#!/bin/bash
# set -x

# A script to copy and check files
# To guard against corruption, the file is chksum'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
```

4.2 A Simple Data Migration Script

```
#!/bin/bash
# set -x

# A script to copy and check files
# To guard against corruption, the file is chksum'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
```

PART V. REFERENCE

CHAPTER V – 1. USER UTILITIES (MAN1)

1.1 lfs

lfs is a Lustre client file system utility that is used to display striping information for file and directories, set striping policy for files and directories, search for files with specific attributes (after the Unix “find” command) and to create or set quotas.

1.1.1 Synopsis

```
lfs

lfs df [-i] [-h] [path]

lfs find [-quiet|-q] [-verbose|-v] [-recursive|-r] <dir/file>

lfs find [–atime|-A N] [–mtime|-M N] [–ctime|-C N] [-maxdepth|-D N] [-print0|-P]

lfs getstripe [-obd|-O <uuid>] [-quiet|-q] [-verbose|-v] \
[-recursive|-r] <dir/file>

lfs setstripe <filename|dirname> <stripe_size> <start_ost> \
<stripe_count>

lfs setstripe -d <dirname>

lfs quotachown [ -i ] <filesystem>

lfs quotacheck [ -ugf ] <filesystem>

lfs quotaon [-ugf] <filesystem>

lfs quotaoff [-ug] <filesystem>

lfs setquota [-u|-g] <name> <block-softlimit> <block-hardlimit> \
<inode-softlimit> <inode-hardlimit> <filesystem>

lfs quota [-o obd_uiid] [-u | -g] <name> <filesystem>

lfs check <mds| osts| servers>

[-print|-p] [-obd|-O <uuid>] <dir/file>

lfs help
```

NOTE: For the above example <filesystem> refers to the mount point of the Lustre file system (Default: /mnt/lustre).

1.1.2 Description

This utility is used to create a new file with a specific striping pattern, determine the default striping pattern, gather the extended attributes (object numbers and location) for a specific file and for setting Lustre quota. It can be invoked interactively without any arguments or in a non-interactive mode.

You can issue the following commands to invoke lfs in an interactive mode.

```
$ lfs

lfs> help
```

To get a complete listing of available commands, type “help” on the lfs prompt. To get basic help on meaning and syntax of a command, type “help command.” The tab key activates command completion. Command history is available via the “UP” and “DOWN” arrow keys.

Here are the sub-commands available:

setstripe:

- creates a new file with a specific striping pattern
- sets the default striping pattern on an existing directory
- deletes the default striping pattern from an existing directory.

getstripe:

- lists the striping pattern for a given file name or files in a given directory
- lists the striping pattern recursively for all files in a directory tree
- lists the files that have objects on a specific OST.

Find: (old usage)

- lists the extended attributes for a given filename or files in a directory
- lists the extended attributes recursively for all files in a directory tree
- lists the files that have objects on a specific OST.

Please note, we have replaced this use of the lfs command by “lfs getstripe.” “lfs find” now matches the traditional UNIX “find.” It will search the directory tree rooted at the given dir/file name for the files that match the given parameters.

Find: (New usage)

--atime (the file was last accessed N*24 hours ago), checks if the file was last accessed, changed, modified N days ago, that is within the interval (N+1,N] days. The number can be specified as +N and -N, for more than and less than N days ago respectively

-- 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 (the file has an object on a specific OST)

--maxdepth allows the find command to descend at most N levels of the directory tree

[--print0|-P] [--print|-p] prints the full file name on the standard output, followed by a null character or a newline respectively.

If one of the options below is specified, lfind works in the so-called “old” mode. This mode is obsolete; use “lfs getstripe” instead. Both “lfs getstripe” and “lfs find” in the “old” mode have the following options:

[--quiet|-q] [--verbose|-v] [--recursive|-r]

> **NOTE**: lfind in the "new" mode can run on a non-Lustre file system, and can cross all the Lustre/non-Lustre and vice versa mount points correctly.

df: reports file system disk space usage or inode usage for each MDS / OST.

quotachown: changes the owner or group of a file on the specified file system.

quotacheck: scans the specified file system for disk usage and creates or updates quota files.

quotaon: turns file system quotas on.

quotaoff: turns file system quotas off.

setquota: sets file system quotas.

quota: displays the disk usage and limits.

check: displays the status of MDS or OSTs (as specified in the command), or all the servers (MDS and OSTs).

osts: lists all the OSTs for the file system.

help: provides brief help on various arguments.

exit/quit: quits the interactive lfs session.

1.1.3 Examples

To create a file striped on one OST:

```
lfs setstripe /mnt/lustre/file1 131072 0 1
```

To create a default striping pattern on an existing directory for all the new files created therein:

```
$ lfs setstripe /mnt/lustre/dir 131072 0 1
```

To delete the default striping pattern on a directory:

```
$ lfs setstripe -d /mnt/lustre/dir
```

(New files will use the default striping pattern created therein.)

stripe size: if you pass a stripe-size of 0, the file system default stripe size will be 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 will be chosen. Otherwise the file will start on the specified OST index (starting at 0).

stripe count: if you pass a stripe-count of 0, the file system default number of OSTs will be 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 the stripe start to 0 all new file creations will occur on OST 0 which is seldom a good idea.

Below is an example of setting and getting stripes:

```
$ 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 extended attributes of a given file:

```
$ lfs find /mnt/lustre/foo1
  OBDS:
     O: OST_localhost_UUID
/mnt/lustre/foo1
obdidx  objid  objid  group
   0      1     0x1     0
```

To list the extended attributes of all files in a given directory:

```
$ lfs find /mnt/lustre/
fs find -r /mnt/lustre/
```

To list all the files that have objects on a specific OST:

```
$ lfs find -r --obd OST2-UUID /mnt/lustre/
```

To change the file owner and group:

```
$ lfs quotachown -i /mnt/lustre
```

To check the quota for a user and a group:

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

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

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

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

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

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

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

To change the owner or group:

```
$ quotachown -i /mnt/lustre
```

To ignore the error if the file does not exist.

For example,

```
$ lfs quotachown -i {file|directory} /mnt/lustre
```

To check the disk space in inodes available on individual MDS and OST:

```
$ 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  244056064    1349   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:

```
$ 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:

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

To check the status of all the servers – MDS and OSTs:

```
$ 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:

```
$ lfs check mds
```

To check the status of all the servers – OSTs:

```
$ lfs check ost
```

To list all the OSTs:

```
$ lfs osts
  OBDS:
     O: OST_localhost_UUID
```

To list the logs of particular types:

```
$ lfs catinfo {keyword} [node name]
```

Keywords are one of the followings: config, deletions.

Node name must be provided when using the keyword config.

For instance,

```
$ lfs catinfo {config|dele*tions}{mdsnode|ostnode}
```

To join the files:

```
 $ join <filename_A> <filename_B>
```

1.2 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 to lmc.

Here are the Lustre-specific options:

Server options: (Currently used by lmc)

extents – Use extended attributes, required

mballoc – Use Lustre filesystem allocator, required

Lustre 1.6 server options:

abort_recov – abort recovery when starting a target (currently an lconf option)

nosvc – start only MGS/MGC servers

exclude – Used to start with a dead OST

Client options:

flock – enable/disable flock support

user_xattr/nouser_xattr – enable/disable user extended attributes

retry= – number of times client will retry mount

CHAPTER V – 2. LUSTRE PROGRAMMING INTERFACES (MAN3)

2.1 Introduction

This chapter describes the public programming interfaces for controlling 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.

2.2 User/Group Cache Upcall

2.2.1 Name

Use **/proc/fs/lustre/mds/mds-service/group_upcall** to look up a given user's group membership.

2.2.2 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 (below) and write it to the **/proc/fs/lustre/mds/mds-service/group_info** pseudo-file.

See **lustre/utils/l_getgroups.c** in the Lustre source distribution for an example upcall program.

2.2.3 Parameters

The name of the MDS service.

The numeric UID.

2.2.4 Data structures

```
#include <lustre/lustre_user.h>

#define MDS_GRP_DOWNCALL_MAGIC 0x6d6dd620

struct mds_grp_downcall_data {
        __u32           mgd_magic;
        __u32           mgd_err;
        __u32           mgd_uid;
        __u32           mgd_gid;
        __u32           mgd_ngroups;
        __u32           mgd_groups[0];
};
```

CHAPTER V – 3. CONFIG FILES AND MODULE PARAMETERS (MAN5)

3.1 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 (Lustre Network Device) modules (for instance, ksocklnd) are loaded automatically by the **lnet** module when LNET starts (typically upon *modprobe ptlrpc*).

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 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 v1.4.6) Lustre configuration lines should be removed from the module configuration files, to be replaced with the following. Make sure that CONFIG_KMOD is set in your linux.config so that LNET can load the following modules it needs. The basic module files are:

- modprobe.conf (Linux 2.6)

```
alias lustre llite
options lnet networks=tcp0,elan0
```

- modules.conf (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 a **Wc** only have effect when connections are established (existing connections are not affected by these changes.)

3.2 Module Options

11. With routed or other multi-network configurations, use *ip2nets* rather than networks so that all nodes can use the same configuration.
12. 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. Keeping a common configuration guarantees that all nodes will have consistent routing tables.
13. A separate modprobe.conf.lnet included from modprobe.conf makes distributing the configuration much easier.
14. If you set "config_on_load=1" LNET starts up at modprobe time, rather than waiting for Lustre to start. This ensures routers start working at module load time. However, in this case **lconf --cleanup** will not stop LNET, you must run **lctl --net stop** on these nodes.
15. Remember **lctl ping** – it is a very handy way to check your LNET configuration.

3.2.1 LNET Options

3.2.1.1 Network Topology

The 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

NOTE: Lustre will ignore the loopback interface (lo0). But Lustre will use any IP addresses aliased to the loopback by default. When in doubt, specify networks explicitly.

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 IP's 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> }

The <net-spec> contains enough information to identify the network uniquely and load an appropriate LND. The LND determines the missing "address-within-network" part of the NID based on the interfaces it can use.

The optional <iface-list> specifies which hardware interface the network can use. If omitted, all the 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 the <iface-list> must be omitted.

The <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, the <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.*. 4 of these nodes also have IP interfaces; these 4 could be used as routers.

Note that match-all expressions (For instance, *.*.*.*) effectively mask all other <net-match> entries specified after them. Hence, they should be used with caution.

Here is a more complicated situation, see below for an explanation of the *route* parameter. We have:

- Two TCP subnets

- One Elan subnet
- One machine set up as a router, with both TCP and Elan interfaces
- We have IP over Elan configured, but IP will only be used to label the nodes.

```
options lnet ip2nets="tcp   198.129.135.* 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  "
```

3.2.1.2 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.

3.2.1.3 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 (that is, 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, 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.

3.2.1.4 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

Acceptor

The acceptor is a TCP/IP service that some LNDs use to establish communications. If a local network requires it and it has not been disabled, the acceptor listens on a single port for connection requests that it redirects to the appropriate local network. The acceptor is part of the LNET module and configured by the following

accept

accept ("secure") is a string that can be set to any of the following values.

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-privledged ports.

none - do not run the acceptor

accept_port

accept_port (988) is the 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

accept_backlog (127) is the maximum length that the queue of pending connections may grow to (see listen(2)).

accept_timeout

accept_timeout (5,W) is the maximum time in seconds the acceptor is allowed to block while communicating with a peer.

accept_proto_version

accept_proto_version is the 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 allows 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 (i.e. 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.

3.2.2 SOCKLND Kernel TCP/IP LND

The 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.

timeout (50,W) is the time in seconds that communications may be stalled before the LND will complete them with failure.

nconnds (4) sets the number of connection daemons.

min_reconnectms (1000,W) is the minimum connection retry interval in milliseconds. This sets the time that must elapse before the first retry after a failed connection attempt. As connections attempts fail, this time is doubled on each successive retry up to a maximum of 'max_reconnectms'.

max_reconnectms (60000,W) is the maximum connection retry interval in milliseconds.

eager_ack (0 on linux, 1 on darwin,W) is a boolean that determines whether the socklnd should attempt to flush sends on message boundaries.

typed_conns (1,Wc) is a boolean that determines whether the socklnd should use different sockets for different types of message. 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) sets the socket buffer sizes. Set this to '0' to allow the system to auto-tune buffer sizes. Be very careful if altering this value as improper sizing can harm the performance.

nagle (0,Wc) is a boolean that determines if nagle should be enabled. It should never be set in production systems.

keepalive_idle (30,Wc) is the time in seconds that a socket can remain idle before a keepalive probe is sent. 0 disables keepalives

keepalive_intvl (2,Wc) is the time in seconds to repeat unanswered keepalive probes. 0 disables keepalives.

keepalive_count (10,Wc) is the number of unanswered keepalive probes before pronouncing socket (hence peer) death.

enable_irq_affinity (1,Wc) is a 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 will disable all zero copy sends. This option is not available on all platforms.

3.2.3 QSW LND

The qswlnd is connectionless, therefore it does not need the acceptor.

It is limited to a single instance, which uses all Elan "rails" that are present and load balances dynamically over them.

The address-with-network is the node's Elan ID. A specific interface cannot be selected in the "networks" module parameter.

tx_maxcontig (1024) is a integer that specifies the maximum message payload in bytes to copy into a pre-mapped transmit buffer.

ntxmsgs (8) is the number of "normal" message descriptors for locally initiated communications that may block for memory (callers block when this pool is exhaused).

nnblk_txmsg (512 with a 4K page size, 256 otherwise) is the number of "reserved" message descriptors for communications that may not block for memory. This pool must be sized large enough so that it is never exhausted.

nrxmsg_small (256) is the number of "small" receive buffers to post (typically everything apart from bulk data).

ep_envelopes_small (2048) is the 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) is the number of "large" receive buffers to post (typically for routed bulk data).

ep_envelopes_large (256) is the number of message envelopes to reserve for the "large" receive buffer queue. See "ep_envelopes_small" above for a further description of message envelopes.

optimized_puts (32768,W) is the smallest non-routed PUT that will be RDMA-ed.

optimized_gets (1,W) is the smallest non-routed GET that will be RDMA-ed.

3.2.4 RapidArray LND

The 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 any 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, the first non-loopback IP interface that is up is used instead.

n_connd (4) sets the number of connection daemons.

min_reconnect_interval (1,W) is the minimum connection retry interval in seconds. This sets the time that must elapse before the first retry after a failed connection attempt. As connections attempts fail, this time is doubled on each successive retry up to a maximum of 'max_reconnect_interval'.

max_reconnect_interval (60,W) is the maximum connection retry interval in seconds.

timeout (30,W) is the time in seconds that communications may be stalled before the LND will complete them with failure

ntx (64) is the number of "normal" message descriptors for locally initiated communications that may block for memory (callers block when this pool is exhaused).

ntx_nblk (256) is the number of "reserved" message descriptors for communications that may not block for memory. This pool must be sized large enough so that it is never exhausted.

fma_cq_size (8192) is the 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) is the size in bytes of the smallest message that will be RDMA-ed, rather than being included as immediate data in an FMA. All messages over 6912 bytes must be RDMA-ed (FMA limit).

3.2.5 VIB LND

The vib lnd is connection based, establishing reliable queue-pairs over Infiniband with its peers. It does not use the acceptor for this.

It is limited to a single instance, which uses a single HCA that can be specified via the "networks" module parameter. It 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.

service_number (0x11b9a2) is the fixed IB service number on which the LND listens for incoming connection requests. Note that all instances of the viblnd on the same network must have the same setting for this parameter.

arp_retries (3,W) is the number of times the LND will retry ARP while it establishes communications with a peer.

min_reconnect_interval (1,W) is the minimum connection retry interval in seconds. This sets the time that must elapse before the first retry after a failed connection attempt. As connections attempts fail, this time is doubled on each successive retry up to a maximum of 'max_reconnect_interval'.

max_reconnect_interval (60,W) is the maximum connection retry interval in seconds.

timeout (50,W) is the time in seconds that communications may be stalled before the LND will complete them with failure.

ntx (32) is the number of "normal" message descriptors for locally initiated communications that may block for memory (callers block when this pool is exhaused).

ntx_nblk (256) is the number of "reserved" message descriptors for communications that may not block for memory. This pool must be sized large enough so that it is never exhausted.

concurrent_peers (1152) is the maximum number of queue pairs, and therefore the maximum number of peers that the instance of the LND may communicate with.

hca_basename ("InfiniHost") is used to construct HCA device names by appending the device number.

ipif_basename ("ipoib") is 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) is a low-level QP parameter. It should not be changed from the default unless advised.

retry_cnt (7,Wc) is a low-level QP parameter. It should not be changed from the default unless advised.

rnr_cnt (6,Wc) is a low-level QP parameter. It should not be changed from the default unless advised.

rnr_nak_timer (0x10,Wc) is a low-level QP parameter. It should not be changed from the default unless advised.

fmr_remaps (1000) controls how often FMR mappings may be reused before they must be unmapped. It should not be changed from the default unless advised.

cksum (0,W) is a boolean that determines whether messages (NB not RDMAs) should be checksummed. This is a diagnostic feature that should not be enabled normally.

3.2.6 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.

n_connd (4) sets the number of connection daemons. The default is 4.

min_reconnect_interval (1,W) is the minimum connection retry interval in seconds. This sets the time that must elapse before the first retry after a failed connection

attempt. As connections attempts fail, this time is doubled on each successive retry up to a maximum of 'max_reconnect_interval'.

max_reconnect_interval (60,W) is the maximum connection retry interval in seconds.

timeout (50,W) is the time in seconds that communications may be stalled before the LND will complete them with failure.

ntx (64) is the number of "normal" message descriptors for locally initiated communications that may block for memory (callers block when this pool is exhausted).

ntx_nblk (256) is the number of "reserved" message descriptors for communications that may not block for memory. This pool must be sized large enough so that it is never exhausted.

concurrent_peers (1024) is the maximum number of queue pairs, and therefore the maximum number of peers that the instance of the LND may communicate with.

cksum (0,W) is a boolean that determines whether messages (NB not RDMAs) should be checksummed. This is a diagnostic feature that should not be enabled normally.

3.2.7 Portals LND (Linux)

The ptllnd can be used as a interface layer to communicate with Sandia Portals networking devices. This version is intended to work on the Cray XT3 Linux nodes using 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 1 message unsolicited. The first message a peer actually sends is a (so-called) "HELLO" message, which is used to negotiate how much additional buffering to set up; typically 8 messages. So if 10000 peers actually exist, we will post enough buffers for 80000 messages.

The maximum message size is set by the *max_msg_size* module parameter (default 512). This parameter sets the bulk transfer breakpoint. Below this breakpoint, payload data is sent in the message itself, and above this breakpoint, a buffer descriptor is sent and the receiver gets the actual payload.

The buffer size is set by the *rxb_npages* module parameter (default 1). The default conservatively avoids allocation problems due to kernel memory fragmentation. However increasing this to 2 is probably not risky.

The ptllnd also keeps an additional *rxb_nspare* buffers (default 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, rising to a max of 10008 as peers actually connected. This could be reduced by a factor of 4 by doubling rxb_npages halving *max_msg_size*.

ME/MD queue length - The ptllnd uses a single portal set by the *portal* module parameter (default 9) for both message and bulk buffers. Message buffers are always attached with PTL_INS_AFTER and match anything sent with "message" matchbits. Bulk buffers are always attached with PTL_INS_BEFORE and match only specific matchbits for that particular bulk transfer.

This scheme assumes that the majority of ME/MDs posted are for "message" buffers, and that the overhead of searching through the preceding "bulk" buffers is acceptable.

Since the number of "bulk" buffers posted at any time is also dependent on the bulk transfer breakpoint set by *max_msg_size*, this seems like an issue worth measuring at scale.

TX descriptors - The ptllnd has a pool of so-called "tx descriptors", which it uses not only for outgoing messages, but also to hold state for bulk transfers requested by incoming messages. This pool should therefore scale with the total number of peers.

To enable the building of the Portals LND (ptllnd.ko) configure with the following option:

./configure --with-portals=<path-to-portals-headers>

ntx (256) The total number of message descriptors

concurrent_peers (1152) The maximum number of concurrent peers. Peers attempting to connect beyond the maximum will not be allowd.

peer_hash_table_size (101) The 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 101. This number should be prime to ensure the peer hash table is populated evenly. Increasing this to 1001 for~10000 peers is advisable.

cksum (0) Set to non-zero to enable message (not RDMA) checksums for outgoing packets. Incoming packets will always be checksumed if necssary, independnt of this value.

timeout (50) The amount of time a request can linger in a peers active queue, before the peer is considered dead. Units: seconds.

portal (9) The portal ID to use for the ptllnd traffic.

rxb_npages (64 * #cpus) The number of pages in a RX Buffer.

credits (128) The maximum total number of concurrent sends that are outstanding at any given instant.

peercredits (8) The maximum number of concurrent sends that are outstanding to a single peer at any given instant.

max_msg_size (512) The maximum immedate message size. This MUST be the same on all nodes in a cluster. A peer connecting with a diffrent max_msg_size will be rejected.

Portals LND (Catamount)

The ptllnd can be used as a interface layer to communicate with Sandia Portals networking devices. This version is intended to work on the Cray XT3 Catamount nodes using Cray Portals as a network transport.

To enable the building of the Portals LND configure with the following option:

./configure --with-portals=<path-to-portals-headers>

The following environment variables can be set to configure the PTLLND's behavior.

PTLLND_PORTAL (9) The portal ID to use for the ptllnd traffic.

PTLLND_PID (9) The virtual pid on which to contact servers.

PTLLND_PEERCREDITS (8) The maximum number of concurrent sends that are outstanding to a single peer at any given instant.

PTLLND_MAX_MESSAGE_SIZE (512) The maximum messages size. This MUST be the same on all nodes in a cluster.

PTLLND_MAX_MSGS_PER_BUFFER (64) The number of messages in a receive buffer. Receive buffer will be allocated of size PTLLND_MAX_MSGS_PER_BUFFER times PTLLND_MAX_MESSAGE_SIZE.

PTLND_MSG_SPARE (256) Additional receive buffers posted to portals.

PTLLND_PEER_HASH_SIZE (101) The number of hash table slots for the peers.

PTLLND_EQ_SIZE (1024) The size of the Portals event queue (that is, maximum number of events in the queue).

CHAPTER V – 4. SYSTEM CONFIGURATION UTILITIES (MAN8)

4.1 mkfs.lustre

mkfs.lustre is a utility for formatting a disk for a Lustre service.

4.1.1 Synopsis

```
mkfs.lustre <target_type> [options] device
```

where <target_type> is one of the following -

--OST object storage target

--MDT meta data storage target

--MGS configuration management service - one per site. This service can be combined with one --mdt service by specifying both types.

4.1.2 Description

mkfs.lustre is used to format a disk device in order to use it as part of a Lustre file system. After formatting, a disk can be mounted to start the Lustre service defined by this command.

OPTIONS

--backfstype=fstype

Force a particular format for the backing fs (like ext3, ldiskfs)

--comment=comment

Set user comment about this disk, ignored by Lustre

--device-size=KB

Set device size for loop devices

--failnode=nid,...

Set the NIDs of a failover partner. This option can be repeated as desired

--fsname=filesystem_name

The Lustre file system of which this service/node will be a part. Default file system name is *lustre*

--index=index

Force a particular OST or MDT index

--mkfsoptions=opts

Format options for the backing fs. For example, ext3 options could be set here

--mountfsoptions=opts

Set permanent mount options, equivalent to the setting in /etc/fstab

--mgsnode=nid,...

Set the NIDs of the MGS node, required for all targets other than the MGS

--noformat

Only print what would be done; this does not affect the disk

--param key=value

Set permanent parameter key to value *value*. This option can be repeated as desired. Typical options might include:

--param sys.timeout=40

System obd timeout

--param lov.stripe.size=2097152

Default stripe size

--param lov.stripe.count=2

Default stripe count

--param failover.mode=failout

Return errors instead of waiting for recovery

--quiet

Print less information

--reformat

Reformat an existing Lustre disk

--stripe-count-hint=stripes

Used for optimizing MDT inode size

--verbose

Print more information.

4.1.3 Examples

To create a file system with MGS and MDT combined on the same node (cfs21) -

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

To create OST for file system testfs on any number of nodes using the above MGS -

```
$ mkfs.lustre --fsname=testfs --ost --mgsnode=cfs21@tcp0 /dev/sdb
```

To create standalone MGS on, say, node cfs22 -

```
$ mkfs.lustre --mgs /dev/sda1
```

To create MDT for file system myfs1 on any node, using the above MGS -

```
$ mkfs.lustre --fsname=myfs1 --mdt --mgsnode=cfs22@tcp0 /dev/sda2
```

4.2 tunefs.lustre

tunefs.lustre is the utility to modify the information of Lustre configuration on a disk.

4.2.1 Synopsis

```
tunefs.lustre [options] device
```

4.2.2 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 file system.

> **WARNING:** Changes made here will affect a file system only when the target is next mounted.

OPTIONS

--comment=comment

Set user comment about this disk, ignored by Lustre.

--erase-params

Remove all previous parameter information.

--failnode=nid, ...

Set the NID(s) of a failover partner. This option can be repeated as desired.

--fsname=filesystem_name

The Lustre file system this service will be part of. Default is 'lustre'.

--index=index

Force a particular OST or MDT index.

--mountfsoptions=opts

Set permanent mount options, equivalent to setting in /etc/fstab.

--mgs

Add a configuration management service to this target.

--mgsnode=nid,...

Set the NID(s) of the MGS node, required for all targets other than the MGS.

--noformat

Only print what would be done; does not affect the disk.

--nomgs

Remove a configuration management service to this target.

--quiet

Print less information.

--verbose

Print more information.

--writeconf

Erase all config logs for the file system that this target is part of. This may prove very dangerous.

4.2.3 Examples

To create a file system with MGS and MDT combined on the same node (cfs21) -

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

To create OST for file system testfs on any number of nodes using the above MGS -

```
$ tunefs.lustre --fsname=testfs --ost --mgsnode=cfs21@tcp0 /dev/sdb
```

To create standalone MGS on, say, node cfs22 -

```
$ tunefs.lustre --mgs /dev/sda1
```

To create MDT for file system myfs1 on any node, using the above MGS -

```
$ tunefs.lustre --fsname=myfs1 --mdt --mgsnode=cfs22@tcp0 /dev/sda2
```

4.3 lctl

lctl is a Lustre utility used for low level configurations of Lustre file system. It also provides low-level testing and manages Lustre network (LNET) information.

4.3.1 Synopsis

```
lctl

lctl --device <devno> <command [args]>

lctl --threads <numthreads> <verbose> <devno> <command [args]>
```

4.3.2 Description

lctl can be invoked in interactive mode by issuing the commands given below.

```
$ 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. To get a complete listing of available commands, type “help” on the lctl prompt. To get basic help on meaning and syntax of a command, type “help command.” Command completion is activated with the TAB key, and command history is available via the “UP” and “DOWN” arrow keys.

For non-interactive single threaded use, one uses the second invocation, which runs command after connecting to the device. Some commands are used only when testing specific functionality inside Lustre and are not normally invoked by users, theses commands are identified by the string (*CFS Dev*). Several commands are old and will be removed in the next major release of Lustre. These commands are identified with the string (*Old*).

Network related options:

--net <tcp/elan/myrinet> The network type to be used for the operation

network <tcp/elans/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 path to a specific host by NID. Can be used to verify network setup and connectivity

add_interface Adds an interface entry (*Old*)

del_interface [ip] Deletes an interface entry (*Old*)

peer_list Displays the peer entries

add_peer <nid> <host> <port> Adds a peer entry (*CFS Dev*)

del_peer [<nid>] [<host>] [ks] Removes a peer entry (*CFS Dev*)

conn_list Displays all the connected remote NIDs

disconnect <nid> Disconnects from a remote NID (*CFS Dev*)

active_tx Displays active transmits, and is used only for elan network type

mynid [nid] Informs the socknal of the local NID. It defaults to host name for tcp networks, and is automatically setup for elan/ myranet networks (*CFS Dev*)

add_uuid <uuid> <nid> Associates a given UUID with an NID (*CFS Dev*)

close_uuid <uuid> Disconnects a UUID

del_uuid <uuid> Deletes a UUID association (*CFS Dev*)

add_route <gateway> <target> [target] Adds an entry to the routing table for the given target (*Old*)

del_route <target> Deletes an entry for a target from the routing table (*Old*)

set_route <gateway> <up/down> [<time>] Enables/ disables routes via the given gateway in the protals 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 (*CFS Dev*)

show_route Displays the complete routing table, same output as **route_list**

ping nid [timeout] [pid] Checks LNET connectivity, outputs a list of NIDs on the target machine

Device Selection:

newdev Creates a new device

device Selects the specified OBD device. All other commands depend on the device being set

**cfg_device <$name>** Sets the current device being configured to <$name> (*Old*)

device_list Shows all the devices

lustre_build_version Displays the Lustre build version

Device Configuration:

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 <args...> 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 file system is on that device

cleanup Cleans up a previously setup device

detach Removes a driver (and its name and UUID) from the current device

lov_getconfig lov-uuid Reads LOV configuration from an MDS device. Returns default-stripe-count, default-stripe-size, offset, pattern, and a list of OST UUIDs (*Old*)

record cfg-uuid-name Records the commands that follow in the log

endrecord Stops recording

parse config-uuid-name Parses the log of recorded commands for a config

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 config log of recorded commands

Device Operations:

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> (*CFS Dev*)

setattr <objid> <mode> Sets the mode attribute for an OST object <objid> (*CFS Dev*)

create [num [mode [verbose]]] Creates the specified number <num> of OST objects with the given <mode> (*CFS Dev*)

destroy <num> Starting at <objid>, destroys <num> number of objects starting from the object with object id <objid> (*CFS Dev*)

test_getattr <num> [verbose [[t]objid]] Does <num> getattrs on an OST object <objid> (objectid+1 on each thread) *(CFS Dev)*

test_brw [t]<num> [write [verbose [npages [[t]objid]]]] Does <num> bulk read/ writes on an OST object <objid> (<npages> per I/O) (*CFS Dev*)

dump_ldlm Dumps all the lock manager states. This is very useful for debugging

activate Activates an import

deacttivate De-activates 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_option 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

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 the catalog logs on 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. It displays all the records from index 1 by default

llog_check <$logname|#oid#ogr#ogen> [from] [to] Checks the log content information. It checks all the records from index 1 by default

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, erases it from the disk

Debug:

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 to plain text format

clear Clears the kernel debug buffer

mark <text> Inserts marker text in the kernel debug buffer

filter <subsystem id/debug mask> Filters message type from the kernel debug buffer

show <subsystem id/debug mask> Shows the specific type of messages

debug_list <subs/types> Lists all the subsystem and debug types

modules <path> Provides gdb-friendly module information

panic Forces the kernel to panic

lwt start/stop [file] Light-weight tracing

memhog <page count> [<gfp flags>] Memory pressure testing

Control:

help Shows a complete list of commands. help <command name> can be used to get help on a specific command

exit Closes the lctl session

quit Closes the lctl session

Options:
(options that can be used to invoke lctl)

--device The device number to be used for the operation. The value of devno is an integer, normally found by calling lctl name2dev on a device name

--threads The numthreads variable is a strictly positive integer indicating the number of threads to be started. The devno option is used as above

--ignore_errors | ignore_errors Ignores errors during the script processing

dump Saves ioctls to a file

4.3.3 Examples

attach

```
$ lctl
lctl > newdev
lctl > attach obdfilter OBDDEV OBDUUID
lctl > dl
 4 AT obdfilter OBDDEV OBDUUID 1
```

getattr

```
lctl > 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
```

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
```

'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
```

4.4 mount.lustre

mount.lustre is a utility to start a Lustre client or target service.

4.4.1 Synopsis

```
$ mount -t lustre [-o options] device dir
```

4.4.2 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) as shown in the section **4.3.1 Synopsis**. 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 file system named <fsname> by contacting the Management Service at <mgsspec>. The format for <mgsspec> is defined below.

<disk_device>

This starts the target service defined by the mkfs.lustre command on the physical disk <disk_device>

OPTIONS

<mgsspec>:=<mgsnode>[:<mgsnode>]

The mgs specification may be a colon-separated list of nodes...

<mgsnode>:=<mgsnid>[,<mgsnid>]

...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:

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:

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)

4.4.3 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
```

CHAPTER V – 5. SYSTEM LIMITS

5.1 Introduction

This section describes various limits on the size of files and file systems. These limits are imposed either by the Lustre architecture or by the Linux VFS and VM subsystems. In a few cases, the limit is defined within the code and could be changed by re-compiling Lustre. In those cases, the limit chosen is supported by CFS testing and may change in future releases.

5.1.1 Maximum Stripe Count

The maximum number of stripe count is 160. This limit is a hard coded option and reflects current tested performance limits. It may be increased in future releases. Under normal circumstances, the stripe count is not affected by ACLs.

5.1.2 Maximum Stripe Size

For a 32-bit machine, the product of stripe size and stripe count (stripe_size * stripe_count) must be less than $2^{\wedge 32}$. The ext3 limit of 2TB for a single file applies for a 64-bit machine. (Lustre can support 160 stripes of 2TB each on a 64-bit system.)

5.1.3 Minimum Stripe Size

Due to the 64KB PAGE_SIZE on some 64-bit machines, the minimum stripe size is set to 64 KB.

5.1.4 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 greater. Rigorous testing is in progress to move the limit to 4000 OSTs.

The maximum number of MDSs will be determined after accomplishing MDS clustering.

5.1.5 Maximum Number of Clients

The number of clients is currently limited to 65536 as defined in the code.

5.1.6 Maximum Size of a File System

In 2.4 kernels, the Linux block layer limits the block devices like hard disks or RAID

arrays to 2TB. For i386 systems in 2.6 kernels, the block devices are limited to 16TB. Each OST or MDS can have a file system up to 2TB (The 2TB limit is imposed by ext3 for 2.6 kernels). You can have multiple OST file systems on a single node. The largest Lustre file system currently has 448 OSTs in a single file system (running the 1.4.3 Lustre version). There is a compile-time limit of 512 OSTs in a single file system, giving a single file system limit of 1PB.

Several production Lustre file systems have around 100 object storage servers in a single file system. One production file system is in excess of 900TB (448 OSTs). All these facts indicate that Lustre would scale just fine if more hardware were made available. The 2TB limit on a file system will be soon removed to allow larger file systems with fewer OST devices.

5.1.7 Maximum File Size

Individual files have a hard limit of nearly 16TB 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 of 2TB. A single file can have a maximum of 160 stripes, which gives an upper single file limit of 320TB 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.

5.1.8 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 2GB depending on the length of the filenames. The maximum number of subdirectories in the versions before Lustre 1.2.6 is 32,000. 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.

5.1.9 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 4KB. That means about 512MB inodes for a file system with MDS of 2TB. This can be increased initially at the time of MDS file system creation by specifying the "--mkfsoptions='-i 2048'" option on the "--add mds" config line for the MDS.

For newer releases of e2fsprogs, you can specify '-i 1024' to create 1 inode for every 1KB disk space. You can also specify '-N {num inodes}' to set a specific number of inodes. Note that 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.

5.1.10 Maximum Length of a Filename and Pathname

This limit is 255 bytes for a single filename, the same as in an ext3 file system. The Linux VFS imposes a full pathname length of 4096 bytes.

Feature List

Supported hardware

Networks

TCP....48

Elan....56

QSW....184

user space tcp

user space portals

Utilities

lfs....167

lfs getstripe....168

lfs setstripe:....168

lfs Find (lfind)....168

lfs check: (lfsck)....169

mount.lustre

mkfs.lustre

tunefs.lustre

lctl....196

Special System Call Behavior

disabling POSIX locking

group locks

Modules....179

LNET....179

Acceptor....182

accept....182

Task List

Key concepts

software

data in /proc

User tasks

free space

change ACL

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.....53
routes.....181

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 that errors can be returned immediately.

C

CFS – Cluster File Systems, Inc., a US corporation founded in 2001 by Peter J. Braam to develop, maintain and support Lustre.

CMD – Clustered meta-data, a collection of meta-data targets implementing a single file system namespace.

CMOBD – Cache Management OBD. A special device which will implement remote cache flushed and migration among devices.

COBD – Caching OBD. A driver which makes decisions when to use a proxy or locally running cache and when to go to a master server. Formerly this abbreviation was used for the word collaborative cache.

Collaborative Cache – A read cache instantiated on nodes that can be clients or dedicated systems, to enable client to client data transfer, enabling enormous scalability benefits for mostly read-only situations. A COBD 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 to that system that a lock it had requested 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 describing 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 describing the default stripe pattern for file underneath that directory.

E

EA – See Extended Attribute.

Eviction – The process of eliminating server state for a client that is not returning to the cluster after a timeout or server failures has occurred.

Export – The state held by a server for a client sufficient to recover all in flight operations transparently when a single failure occurs.

Extended attribute – A small amount of data which can be retrieved through a name associated with a particular inode. Examples of Extended Attributes are access control lists, striping information and crypto keys.

Extent Lock – A lock used by the OSC to protect an extent in a storage object for concurrency control of read, write, file size acquisition and truncation operations.

F

Failback – The failover process whereby the default active server regains control over the service.

Failout OST – An OST which when fails to answer client requests is not expected to recover. A failout OST which has failed can be administratively failed, enabling clients to return errors when accessing data on the failed OST without making network requests.

Failover – The process whereby a standby computer server system takes over for an active computers server after a failure of the active node, typically gaining exclusive access to a shared storage device between the two servers.

FID – A Lustre file identifier. A collection of integers which uniquely identify a file or object. The structure contains a sequence, identity and version number.

Fileset –

FLDB – FID Location Database. This database maps a sequence of FID's to a server which is managing the objects in the sequence.

Flight Group – A 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 system that an extent lock it is holding should be surrendered if it is not in use. If the lock is in use 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

Group Lock –

Group upcall –

GSS API –

H

Htree – An indexing system for large directories used by ext3. Originally implemented by Daniel Phillips and completed by CFS.

I

Import – The state held by a client to recover a transaction sequence fully 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 result of the operation without granting a lock. The use of intent locks leads to even complicated meta-data operations implemented with a single RPC from the client to the server.

IOV – IO vector. A buffer destined for transport across the network which contains a collection, aka as a vector, of blocks with data.

J

Join File –

K

Kerberos – An authentication mechanism, optionally available in 1.6 versions of Lustre as a GSS backend.

L

LAID – Lustre RAID. A mechanism whereby the LOV can stripe I/O over a number of OST's with redundancy. Expected in Lustre 2.0.

LBUG – A bug written into a log by Lustre indicating a serious failure of the system.

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), ifs (Intermezzo).

Lfsck – Lustre File System Check - a distributed version of a disk file system checker. Lfsck normally does not need to be run, except when file systems incurred damage through multiple disk failures and other forms of damage that cannot be recovered with 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, don't need to give back locks on time, and can recover safely from an eviction. They should not participate in recovery.

Llite – See Lustre Lite. The word is still in use inside the code and module names to indicate that code elements are related to the Lustre file system.

Llog – A log file of entries used internally by Lustre. An llog is suitable for rapid transactional appending of records and very cheap cancellation of records through a bitmap.

Llog 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 records, usually through an RPC, when the records are not needed.

LMV – Logical meta-data volume, a driver to abstract in the Lustre client that it is working with a meta-data cluster instead of a single meta-data server.

LND – Lustre Network Driver, a code module enabling LNET support over a particular transport, such as TCP, 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 RPC protocol deal with stateful servers and has exactly-once semantics, and built in support for recovery.

Load Balancing MDS – A cluster of MDS's that perform load balancing of the requests among the systems.

Lock Client – A module making lock RPC's to a lock server and handling revocations from the server.

Lock Server – A system managing locks on certain objects. It also issues lock callback requests calls while servicing or completing lock requests for already locked objects.

LOV – Logical object volume. This is the object storage analog of a logical volume in a block device volume management system such as LVM or EVMS. The logical object volume is primarily used to present a collection of OST's as a single object 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 OST's.

LOV Logical Object Volume – An OBD providing access to multiple OSC's and presenting the combined result as a single device.

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 meta-data server which contains references to storage object on OSS servers.

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 providing an interface between Lustre OSD and MDD drivers and file systems, to avoid introducing file system specific abstractions into the OSD and MDD drivers.

M

Mballoc – An advanced block allocation protocol introduced by CFS into the ext3 disk file system capable of efficiently managing the allocation of large (typically 1MB) contiguous disk extents.

MDC – The meta-data client code module which interacts with the MDT using LNETrpc. Also an instance of an object device operating on an MDT through the network protocol.

MDD – A meta-data device, currently implemented using the directory structure and extended attributes of disk filesystems.

MDS – Meta-data Server, referring to a computer system or software package running the Lustre meta-data services.

MDS Client – Same as MDC.

MDS Server – Same as MDS.

MDT – A meta-data target, a meta-data device made available through the Lustre meta-data network protocol.

Meta-data Writeback Cache – Many local and network filesystems have a cache of file data which applications have written but which has not yet been flushed to storage devices. A meta-data writeback cache is a cache of

meta-data updates (mkdir, create, setattr, other operations) which an application has performed and which have not yet been flushed to a storage device or server. InterMezzo is one of the first network filesystems to have a meta-data write back cache.

MGS – Management service. A software module managing startup configuration information and changes to this information. Also the server node on which this system is running.

Mount object –

Mountconf – The configuration protocol for Lustre introduced in version 1.6 where formatting disk file systems on servers with the mkfs.lustre program prepares them for automatic incorporation into a Lustre cluster.

N

NAL – An older, obsolete term for LND.

NID – A network id, which 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 implementing a library for sending large network requests, moving buffers with RDMA.

O

OBD – Object device, the base class of layering software constructs that provides the Lustre functionality.

OBD API – See storage object API.

OBD type– Many modules can implement the Lustre object or meta-data API's. Examples of OBD types are the LOV, the OSC and the OSD.

Obdfilter – An older name for the OSD device driver.

OBDFS Object Based File System – A now obsolete single node object filesystem storing data and meta-data on object devices.

Object device – An instance of a object that exports the OBD API.

Object storage – A term referring 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 (XXX supply numbers of standards here) and the Lustre object storage protocol. The Lustre protocol is 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 – cache of open file handles. Performance enhancement for NFS

Orphan objects – Storage objects for which there is no Lustre file pointing anymore at these objects. 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 tells the OST to delete the stripe. The OST finally sends the cookie back to the MDT to cancel it.

Orphan handling – A component of the meta-data service which allows for recovery of open unlinked files after a server crash. The implementation of this features retains open unlinked files as orphan objects until it is clear that no clients are using them.

OSC Object Storage Client – The client unit talking to an OST (via an OSS).

OSD – Object Storage Device. This term is a generic term used in the industry for storage devices with a more extended interface than block oriented devices such as disks. Lustre uses this name for a software module

implementing an object storage API in the kernel. Lustre also uses this name for an instance of an object storage device created by that driver. The OSD device is layered on a file system, with methods that mimic the creation, destroy and I/O operations on file inodes.

OSS – Object Storage Server. A system running an object storage service software stack.

OSS Object Storage Server – A server OBD providing access to local OST's.

OST – Object storage target, an object storage device made accessible through a network protocol. An OST is typically tied to a unique OSD which in turn is tied to 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 OST's can be combined into a pool with unique access permissions and stripe characteristics. Each OST is a member of only 1 pool, while an MDT can serve files from multiple pools. A client accesses one pool on the the filesystem; the MDT stores files from/for that client only on that pool's OST's

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 by as integers.

Examples of portals are the portals on which certain groups of object, meta-data, 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 –

Replay – The concept of re-executing a request on a server after a server shutdown where the server lost information in its memory caches. The requests to be replayed 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.

Resent request – Requests that have 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 notion 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 at least one client system should not be subject to root squash for management purposes.

routing – LNET can route between different networks and LNDs

RPC – Remote procedure call, a network encoding of a request**.**

S

Storage Object API – The API manipulating storage objects. This API is richer than that of block devices and includes the creation and deletion of storage objects, reading and writing buffers from/to certain offsets, setting attributes and other storage object meta-data.

Storage objects – A generic notion referring to data containers, similar or identical to file inodes.

Stride – A contiguous logical extent of a Lustre file written to a single object service target.

Stride size – The maximum size of a stride, typically 4MB.

Stripe count – The number of OST's holding objects for a RAID0 striped Lustre file.

Striping meta-data – The extended attribute associated with a file describing how its data is distributed over storage objects. See also default stripe pattern, and directory striping meta-data.

T

T10 object protocol – An object storage protocol tied to the SCSI transport layer.

W

Wide striping – Using many OST's to store stripes of a single file to obtain maximum bandwidth to a single file through parallel utilization of many OST's.

Z

zeroconf – Obsolete from 1.6. A method to start a client without an XML file. The mount command gets a client startup llog from a specified MDS.

ALPHABETICAL INDEX

P

Q

R

S

T

V

X

Version Log

Version No.	Details of the changes made	Author	Date
1.1	1) Upgraded all the chapters from 1.4 to 1.6 version of Lustre 2) Introduction and Information of new features of Lustre 1.6 like MountConf, MGS, MGC, and so on 3) Introduction and information of utilities like mkfs.lustre, mount.lustre and tunefs.lustre 4) Removed lmc and lconf utilities 5) Added Chapter II – 10. Upgrading Lustre from 1.4 to 1.6 6) Removed the Appendix Upgrading from 1.4.5 to 1.4.6 7) Added information on how to remove an OST permanently	SPSOFT	02/03/07