Lustre* Installation and Configuration using Intel® EE for Lustre* Software and OpenZFS

A system integrator's guide to the Lustre parallel file system

High Performance Data Division

June 7, 2016
World Wide Web: http://www.intel.com

Disclaimer and legal information

The source code contained or described herein and all documents related to the source code ("Material") are owned by Intel® Corporation or its suppliers or licensors. Title to the Material remains with Intel® Corporation or its suppliers and licensors. The Material contains trade secrets and proprietary and confidential information of Intel® or its suppliers and licensors. The Material is protected by worldwide copyright and trade secret laws and treaty provisions. No part of the Material may be used, copied, reproduced, modified, published, uploaded, posted, transmitted, distributed, or disclosed in any way without Intel's prior express written permission.

No license under any patent, copyright, trade secret or other intellectual property right is granted to or conferred upon you by disclosure or delivery of the Materials, either expressly, by implication, inducement, estoppel or otherwise. Any license under such intellectual property rights must be express and approved by Intel® in writing.

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL® ASSUMES NO LIABILITY WHATSOEVER AND INTEL® DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL® PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

A "Mission Critical Application" is any application in which failure of the Intel® Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL'S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL® AND ITS SUBSIDIARIES, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS' FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL® OR ITS SUBCONTRACTOR WAS NEGLIGENT IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL® PRODUCT OR ANY OF ITS PARTS.

Intel® may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined". Intel® reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Before using any third party software referenced herein, please see the third party software provider's website for more information, including without limitation, information regarding the mitigation of potential security vulnerabilities in the third party software.

Contact your local Intel® sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel® literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm.

Intel® and the Intel® logo are trademarks of Intel® Corporation in the U.S. and/or other countries.

* Other names and brands may be claimed as the property of others.

"This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit. (http://www.openssl.org/)

Contents

About this Document

Conventions Used

Conventions used in this document include:

- # preceding a command indicates the command is to be entered as root
- $ indicates a command is to be entered as a user
- *<variable_name>* indicates the placeholder text that appears between the angle brackets is to be replaced with an appropriate value

Related Documentation

- *Intel® Enterprise Edition for Lustre* Software Installation Guide*
- *Intel® Manager for Lustre* Software User Guide*
- *Hierarchical Storage Management Configuration Guide*
- *Installing Hadoop, the Hadoop Adapter for Intel® EE for Lustre*, and the Job Scheduler Integration*
- *Creating an HBase Cluster and Integrating Hive on an Intel® EE for Lustre*® File System*
- *Creating a High-Availability Lustre* Storage Solution over a ZFS File System*
- *Upgrading a Lustre file system to Intel® Enterprise Edition for Lustre* Software (Lustre only)*
- *Configuring LNet Routers for File Systems based on Intel* EE for Lustre* Software*
- *Configuring SELinux for File Systems based on Intel* EE for Lustre* Software*
- *Creating a Scalable File Service for Windows Networks using Intel® EE for Lustre* Software*
- *Intel® EE for Lustre* Hierarchical Storage Management Framework White Paper*
- *Architecting a High-Performance Storage System White Paper*

Introduction to Lustre*

Overview

Lustre* is a global single-namespace, POSIX-compliant, distributed parallel file system architecture designed for scalability, high-performance, and high-availability. Lustre runs on Linux-based operating systems and employs a client-server network architecture. Storage is provided by a set of servers that can scale to populations measuring up to several hundred hosts. Lustre servers for a single file system instance can, in aggregate, present up to tens of petabytes of storage to thousands of compute clients, with more than a terabyte-per-second of combined throughput.

Lustre is a file system that scales to meet the requirements of applications running on a range of systems from small-scale HPC environments up to the very largest supercomputers and has been created using object-based storage building blocks to maximize scalability.

Redundant servers support storage fail-over, while metadata and data are stored on separate servers, allowing each file system to be optimized for different workloads. Lustre can deliver fast IO to applications across high-speed network fabrics, such as Intel® Omni-Path Architecture (OPA), InfiniBand* and Ethernet.

Architecture

The Lustre file system architecture is designed as a scalable storage platform for computer networks and is based on distributed, object-based storage. The namespace hierarchy is stored separately from a file's content. Services in Lustre are separated into those supporting metadata operations, and those supported file content operations.

There are two object types in Lustre. Data objects are simple arrays used to store the bulk data associated with a file's content, while index objects are used to store key-value information, such as POSIX directories. Block storage management is delegated to back-end storage servers and all application-level file system access is transacted on a network fabric between clients and the storage servers.

The building blocks of a Lustre file system are the Metadata Servers and Object Storage Servers, which provide namespace operations and bulk IO services respectively. There is also the Management Server, a global resource that is functionally independent of any single Lustre instance and the clients that present a coherent, POSIX interface to end-user applications. Lustre's network protocol, LNet, joins all of the components into a seamless whole.

Metadata Server

The Metadata Server (MDS) manages all name space operations for a Lustre file system. A file system's directory hierarchy and file information are contained on a storage device referred to as a Metadata Target (MDT), and the MDS provides the logical interface to this storage. A Lustre file system will always have at least one MDS and corresponding MDT, and more can be added to meet the scaling requirements of a particular environment. One metadata server (MDS) can have one or more metadata targets (MDT), and there can be more than one metadata server per Lustre file system. The MDS controls the allocation of storage objects for the file content when a file is created, and manages the opening and closing of files, file deletions and renames, and other namespace operations. The MDS does not participate in I/O after a file is opened, until it is time to close the file.

An MDT stores namespace metadata, such as filenames, directories, access permissions, and file layout, effectively providing the index for the data held on the file system. The MDT data is stored in a direct-attached disk storage system. The ability to have multiple MDTs in a single file system allows directory subtrees to reside on the secondary MDTs, which is useful for isolating workloads that are especially metadata-intensive onto dedicated hardware (one could allocate an MDT for a specific set of projects, for example). Large, single directories can be distributed across multiple MDTs as well, providing scalability for applications that generate large numbers of files in a flat directory hierarchy.

Management Server

The Management Server (MGS) stores configuration information for all the Lustre file systems in a cluster and provides this information to other Lustre components. Each Lustre target contacts the MGS to provide information, and Lustre clients contact the MGS to retrieve information. The MGS can be paired with an MDS in a high-availability configuration, with each server connected to shared storage. Multiple Lustre file systems can be managed by a single MGS.

Object Storage Server

The Object Storage Servers provide bulk storage for the contents of files in a Lustre file system. One or more object storage servers (OSS) store file data on one or more object storage targets (OST). A single OSS typically serves between two and eight OSTs (although more are possible), with the OSTs stored on direct-attached storage. The capacity of a Lustre file system is the sum of the capacities provided by the OSTs. OSSs are usually paired, with each OST accessible via two servers, to provide failover in the event of a server or component failure. Paired servers support improved throughput during normal operation and high-availability of the file system.

Clients

Applications access and use file system data by interfacing with a Lustre client. Lustre clients present applications with a unified namespace for all of the files and data in the file system, using standard POSIX semantics. A Lustre file system is mounted on the client operating system just as for any other POSIX file system; each Lustre instance is presented as a separate mount point on the client's operating system, and each client can mount several different Lustre file system instances concurrently.

Networking

LNet is the high-speed data network protocol that clients use to access the file system. LNet is designed to meet the needs of large-scale compute clusters and is optimized for very large node counts and high throughput. LNet supports Ethernet, InfiniBand*, Intel® Omni-Path Architecture (OPA), and specific compute fabrics such as as Cray* Gemini.

LNet abstracts network details from the file system itself. LNet allows for full RDMA throughput and zero copy communications when available.

LNet supports routing, which provides maximum flexibility for connecting different network topologies. LNet routing provides an efficient protocol for bridging different networks, or employing different fabric technologies, such as Intel® OPA and InfiniBand. In larger file systems, with very large client counts, LNet can also support multihoming to improve performance and reliability. For more details and guidance, see the document Configuring LNet Routers for File Systems based on Intel* EE for Lustre* Software.

High Availability and Data Storage Reliability

Lustre Storage

Organizations must be able to trust in the reliability and availability of their IT infrastructure resources if they are to be successful in the realization of their objectives. For storage systems, this means that users must have confidence that their data is stored persistently and reliably without loss or corruption of information, and that the data, once stored, is available for recall on demand to an application.

Lustre is designed to keep up with the demands of the most data-intensive workloads from applications running on the very largest supercomputers in the world. Any overheads that are introduced into these environments reduces the usable bandwidth for applications and reduces overall efficiency, which in turn increases the time it takes to arrive at a result. Therefore, the Lustre file system architecture does not implement a redundant storage pattern for data objects across storage servers, due to the inherent latency and bandwidth overheads that replication and other data redundancy mechanisms introduce.

Data reliability is implemented in the storage subsystem, where it can be isolated in large part from user application-level I/O and communications. These storage systems typically comprise multi-ported enclosures, each containing an array of disks or other persistent storage devices. Arrays may be intelligent data storage systems with dedicated controllers or simple trays with no dedicated control software (usually referred to as JBODs, "Just a Bunch of Disks").

Intelligent storage arrays have the benefit of abstracting the complexity of managing storage redundancy through RAID configuration, and offloading the computational overheads for checksum and parity calculations from the host server. Dedicated storage controllers are typically configured with battery-backed cache for buffering data, thereby further isolating from the storage services running on the host computer the IO overhead associated with writing additional data blocks (e.g. in a RAID 6 or RAID 10 device layout).

JBOD enclosures are simpler and are less expensive than intelligent storage arrays (often referred to as "hardware RAID arrays"). The low cost of acquisition is offset by more complex software configuration in the host server's operating system, as all data management tasks, including redundant layout configuration, must be performed and monitored by the host. The Linux kernel's standard tools for storage volume management, MDRAID and LVM, provide the basic tools for managing JBOD storage and allow for complex fault tolerant disk layouts to be defined. After the block layout is defined, a file system can be formatted on top of the software volume.

LVM and MDRAID, while prevalent, are somewhat complex and can be difficult to manage efficiently on large-scale storage platforms. However in recent years, the JBOD architecture has received a significant boost in popularity thanks to the development of advanced file system technology, as exemplified by OpenZFS, which makes storage management easier while at the same time improving reliability and data integrity. OpenZFS has disrupted many of the assumptions regarding storage management and "software RAID" since its original introduction in the Solaris operating system by Sun Microsystems (now Oracle Solaris).

The OpenZFS file system reduces the administrative complexity of maintaining software-based storage by taking a holistic view of both the file system and storage management. ZFS integrates volume management features with an advanced file system that scales efficiently and provides enhancements including end-to-end checksums for protection against data corruption, versatility in storage configuration, online data integrity verification, and a copy-on-write architecture that eliminates the need to perform offline repairs. There is no `fsck` in ZFS.

Advances in software-based storage architectures are also influencing storage hardware design, creating hybrid server and storage enclosures that combine storage trays with standard servers into a single high-density chassis. These integrated systems can offer higher density per rack and less complex physical integration (including reduced cabling).

High Availability for Lustre Service Continuity

Lustre servers are responsible for transacting I/O requests from applications running on a network of computers, and for managing the block storage used to maintain a persistent record of the data. Lustre clients do not have direct connections to the block storage and are often completely diskless, with no local data persistence. Because data is not replicated between Lustre servers, loss of access to a server means loss of access to the data managed by that server, which in turn means that a subset of the data managed by the Lustre file system will not be available to clients.

To protect against service failure, Lustre data is usually held on multi-ported, dedicated storage to which two or more servers are connected. The storage is subdivided into volumes or LUNs, with each LUN representing a Lustre storage target (MGT, MDT or OST). Each server that is attached to the enclosure has equal access to the storage targets and can be configured to present the storage targets to the network, although only one server is permitted to access an individual storage target in the enclosure at any given time. Lustre uses an inter-node failover model for maintaining service availability, meaning that if a server develops a fault, then any Lustre storage target managed by the failed server can be transferred to a surviving server that is connected to the same storage array.

This configuration is usually referred to as a high-availability cluster. A single Lustre file system installation will be comprised of several such HA clusters, each providing a discrete set of services that is a subset of the whole. These discrete HA clusters are the building blocks for a high-availability, Lustre parallel distributed file system that can scale to tens of petabytes in capacity and to more than one terabyte-per-second in aggregate throughput performance.

Building block patterns can vary, which is a reflection the flexibility that Lustre affords integrators and administrators when designing their high performance storage infrastructure. The most common blueprint employs two servers joined to shared storage in an HA clustered pair topology. While HA clusters can vary in the number of servers, a two-node configuration provides the greatest overall flexibility as it represents the smallest storage building block that also provides high availability. Each building block has a well-defined capacity and measured throughput, so Lustre file systems can be designed in terms of the number of building blocks that are required to meet capacity and performance objectives.

A single Lustre file system can scale linearly based on the number of building blocks. The minimum HA configuration for Lustre is a metadata and management building block that provides the MDS and MGS services, plus a single object storage building block for the OSS services. Using these basic units, one can create file systems with hundreds of OSSs as well as several MDSs, using HA building blocks to provide a reliable, high-performance platform.

Figure 1 shows a blue-print for a typical two-node, high-availability Lustre server building block.

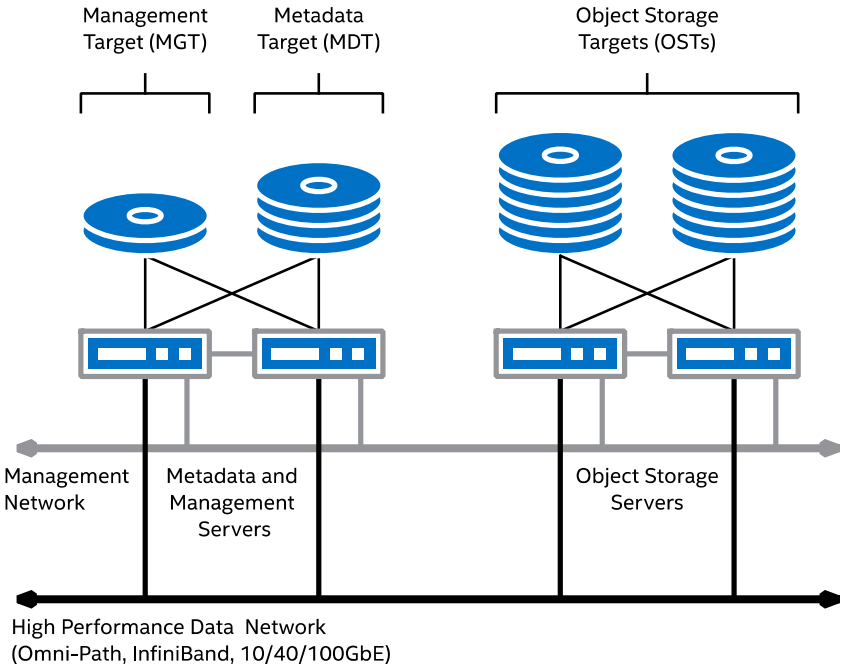

Figure 1. Lustre Server High-Availability Building Blocks

Lustre doesn't need to be configured for high availability – a Lustre file system will operate perfectly well without HA protection, but be aware that a fault in the server infrastructure will cause a service outage for the file system. Designing Lustre for high availability is therefore recommended, and is the norm for the overwhelming majority of Lustre file system installations.

High Availability and GNU/Linux

Every major enterprise operating system offers a high-availability cluster software framework that follows this basic model. In current[1] GNU/Linux distributions, the de facto HA cluster framework has consolidated around two software packages: *Corosync* for cluster membership and communications, and *Pacemaker* for resource management.

Operating system distributions have each developed their own administration tools around these applications. For example, Red Hat Enterprise Linux (RHEL) makes use of PCS[2] (Pacemaker/Corosync Configuration System), while SuSE Linux Enterprise Server (SLES) has CRMSH[3] (Cluster Resource Management Shell). Both PCS and CRMSH are open-source

[1] Current as of 2016
[2] https://github.com/feist/pcs
[3] http://crmsh.github.io

applications. There are also a number of other tools available, including a web-based application HAWK[4], which interfaces to CRMSH. PCS has its own web-based UI.

While this diversity of tools has the effect of limiting portability of the specific procedures for creating and maintaining clusters, the underlying technology remains the same.

Throughout the remainder of this text, Red Hat Enterprise Linux (RHEL) will be used as the reference operating system platform, along with the PCS command line interface for HA software configuration.

RHEL has a complex legacy of software tools for creating HA frameworks, but with the RHEL 6.4 release, and the release of RHEL 7, the operating platform has been simplified and streamlined. RHEL 6 still has some legacy software infrastructure, being based on older versions of Corosync and Pacemaker than RHEL 7, but for the most part this is hidden from the systems administrator by PCS. Where there are differences in procedure between RHEL 6 and 7, they will be identified herein.

[4] https://github.com/ClusterLabs/hawk

Lustre Reference Architecture in this Guide

Overview

Figure 2 depicts the overall architecture of a typical Lustre file system installation. Lustre can support up to 32 Metadata servers (MDS) and up to 2000 Object Storage servers (OSS) per file system. This guide will focus on file systems that require only a single MDS, which represents the majority of Lustre installations in production today.

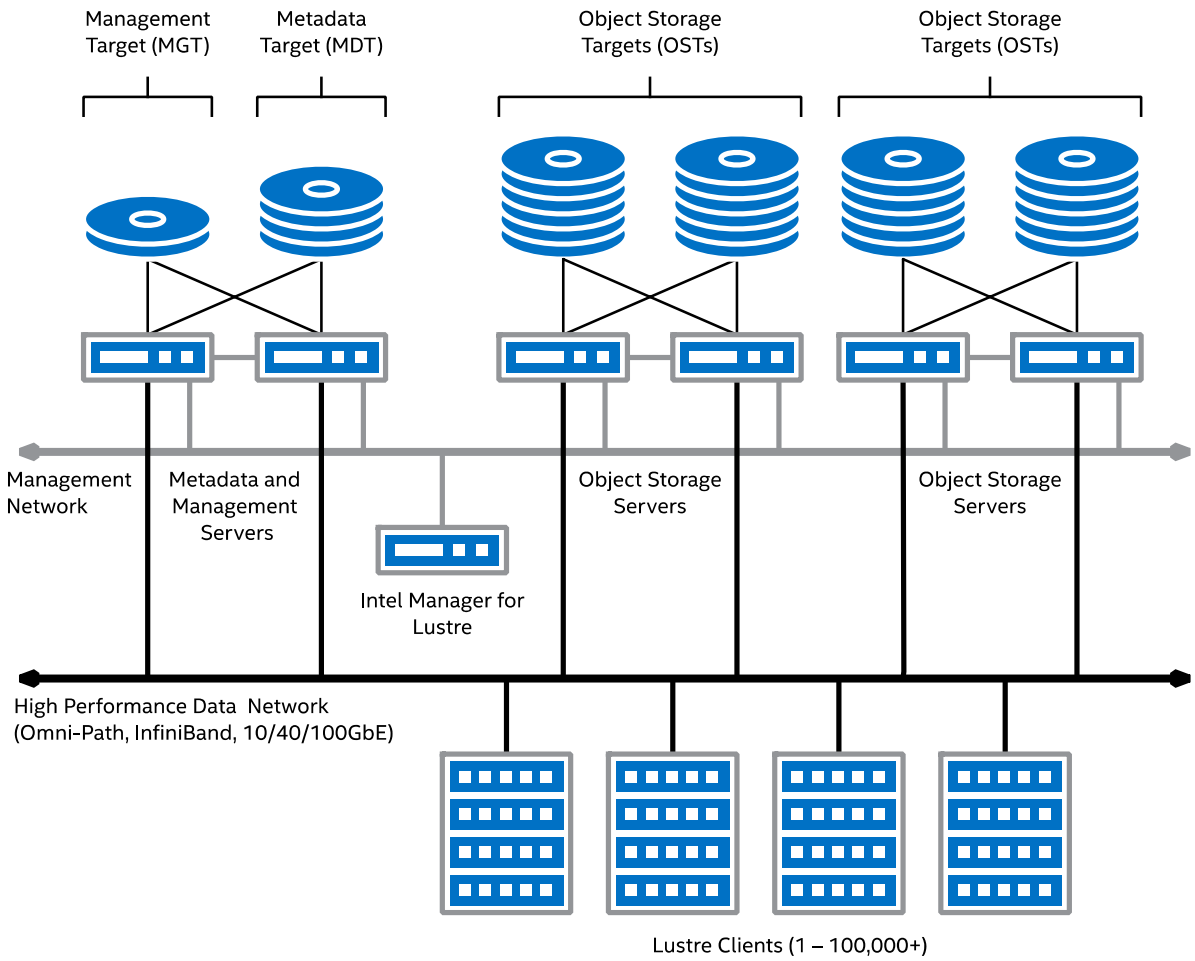

Figure 2. Lustre High-Availability Network Architecture

The guide will use tested examples based on this basic system architecture:

- one administration server from which to coordinate tasks
- two metadata servers, each equipped with:

  - three network interface cards
  - one storage enclosure accessible from both hosts to contain the MGT and MDT0
- two object storage servers (HA pair #1), each equipped with:
  - three network interface cards
  - one storage enclosure accessible from both hosts to contain OST0 and OST1
- two object storage servers (HA pair #2), each equipped with:
  - three network interface cards
  - one enclosure accessible from both hosts to contain OST2 and OST3
- four Lustre clients
- one management network
- one high performance data network

Network

The examples in this guide will for the most part use Ethernet networking for both the management and high performance data networks. However, it should be noted that it is common for Lustre file systems to be incorporated onto high throughput, low-latency fabrics such as Intel® OPA and InfiniBand, and the Lustre Networking protocol has drivers specifically developed to exploit the performance available in these fabrics.

Software

Operating System

The reference architecture in this guide has been developed on Red Hat Enterprise Linux and covers the RHEL 6.x and 7.x distributions, as well as CentOS. Configuration of SuSE Linux is not within the scope of the guide.

YUM is used extensively in the reference architecture examples to manage the installation of software. The documentation examples herein assume that access is available to RPM package repositories on Red Hat Network (RHN) or via Red Hat Satellite (RHS), or the CentOS updates repositories.

If using RHEL, an active subscription is required for the HA add-on software in RHN.

The Lustre binary packages are created for very specific versions of the operating system's Linux kernel. This does vary over time, as OS distribution vendors such as Red Hat release periodic updates to the kernel packages. Prior to commencing an installation of Lustre, be sure to download the latest Lustre software distribution to ensure the best compatibility with the most current OS update.

The MGS, MDS and OSS servers as well as the Lustre clients can use the same core operating platform and can be installed from a common template. A sample Lustre kick start template has been included in Appendix X for reference, and is suitable for both RHEL 6 and RHEL 7 based servers and clients.

It is common practice to align the operating system installations for Lustre servers as closely as possible. Ideally, each server will have the same core operating system environment installed from a common template, and the software package manifests and versions will match across all of the server assets. Variation of software and hardware is discouraged, but can occur when the deployment has been in place for some years and inevitable changes in hardware catalogues mandate a change in the design of individual servers.

In high availability configurations, each server of the HA group should ideally be identical, or as similar as possible in every design aspect, from server hardware to slot placement for cards, CPU choices and memory configuration. Each server should have an identical software package manifest to ensure consistent performance and application behavior during operation of the cluster group.

Lustre clients tend to be more heterogeneous, with a wider variety of operating systems and hardware configurations. This is especially true in data centers where the Lustre file system is a resource shared across multiple HPC clusters, or is a globally-deployed asset within an organization. While keeping the Lustre servers and clients aligned is generally recommended, it is nevertheless quite common for the Lustre servers to differ from Lustre clients in OS distributions and Lustre versions.

Intel® Enterprise Edition for Lustre* Software

The examples in this guide will use the Intel® Enterprise Edition for Lustre* software distribution, although the principals and practices apply equally to all Lustre distributions.

For information on the Intel's Lustre solutions portfolio, including Intel® Enterprise Edition for Lustre software, visit www.intel.com/lustre.

Metadata and Management Servers

Figure 3 below shows a typical blue print for the Metadata server high availability building block. The building block comprises two servers, connected to a common external storage enclosure. The storage array in the diagram has been configured to hold the MGT as well as MDT0 for a Lustre file system. Two of the drives have been retained as spares. Each server has some internal, node-local storage to host the operating system, typically two disks in a RAID 1 mirror. There are three network interfaces on each server: a high performance data network for Lustre and other application traffic, a management network for administration of the servers, and a dedicated point-to-point server connection for use as a communication ring in high availability framework, such as might be provided by a combination of Pacemaker and

Corosync. Corosync is used for communications in HA clusters, and can be configured to use multiple networks for its communications rings. In this reference topology, both the dedicated point-to-point connection and the management network can be used in Corosync rings. Pacemaker high-availability clusters also commonly include a connection to a power management interface (not pictured), which is used to isolate, or fence, machines when a fault is detected in the cluster. Examples of the power management interface include a network or serial device connection to a smart PDU, or to a server's BMC via IPMI. The management network is sometimes used to provide the connection to these power management interfaces.

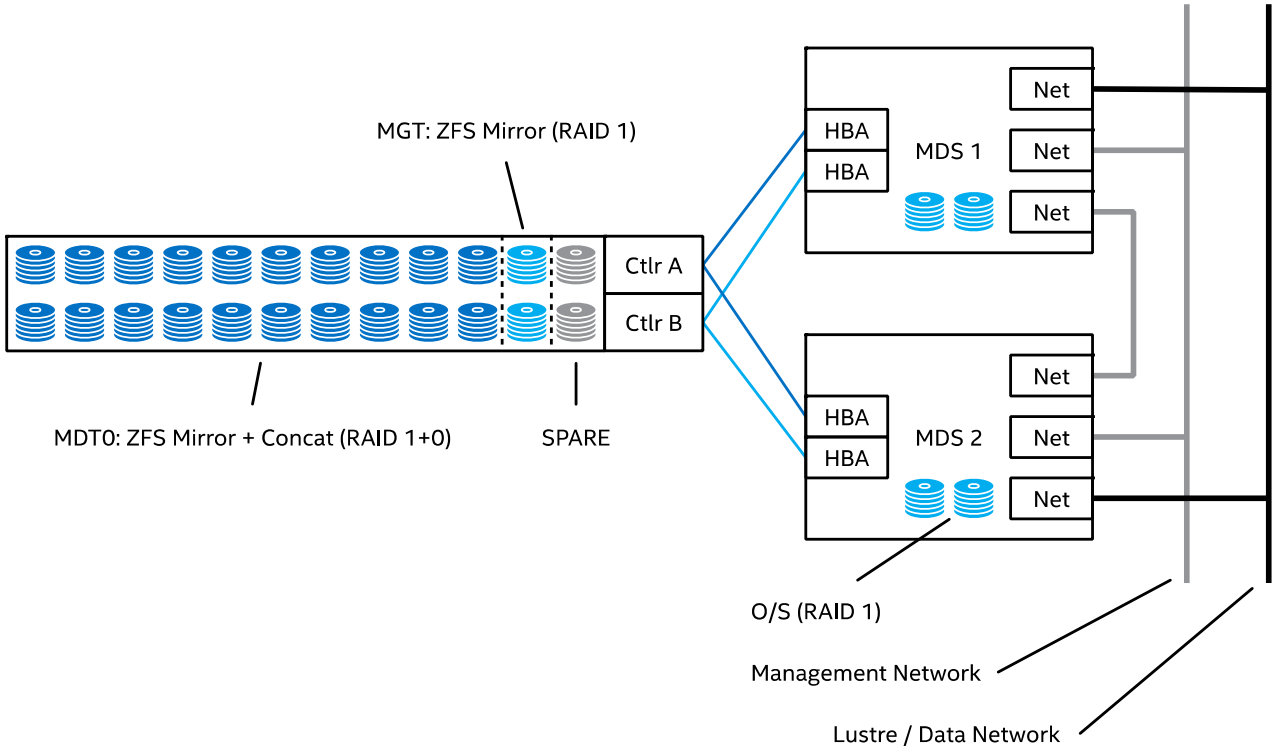

Figure 3. MGS and MDS (MDT0) HA Cluster Building Block

Object Storage Servers

The physical structure of a typical OSS building block is very similar to that of the MDS. The most significant external difference is the storage configuration, which is typically designed to maximize capacity and read/write bandwidth. Storage volumes are commonly formatted for RAID 6, or in the case of ZFS, RAID-Z2. This provides a balance between optimal capacity and data integrity. The example in Figure 4 is a low-density configuration with only 48 disks across two enclosures, and it is not uncommon to see much higher density storage enclosures with 60 disks or more attached to object storage servers.

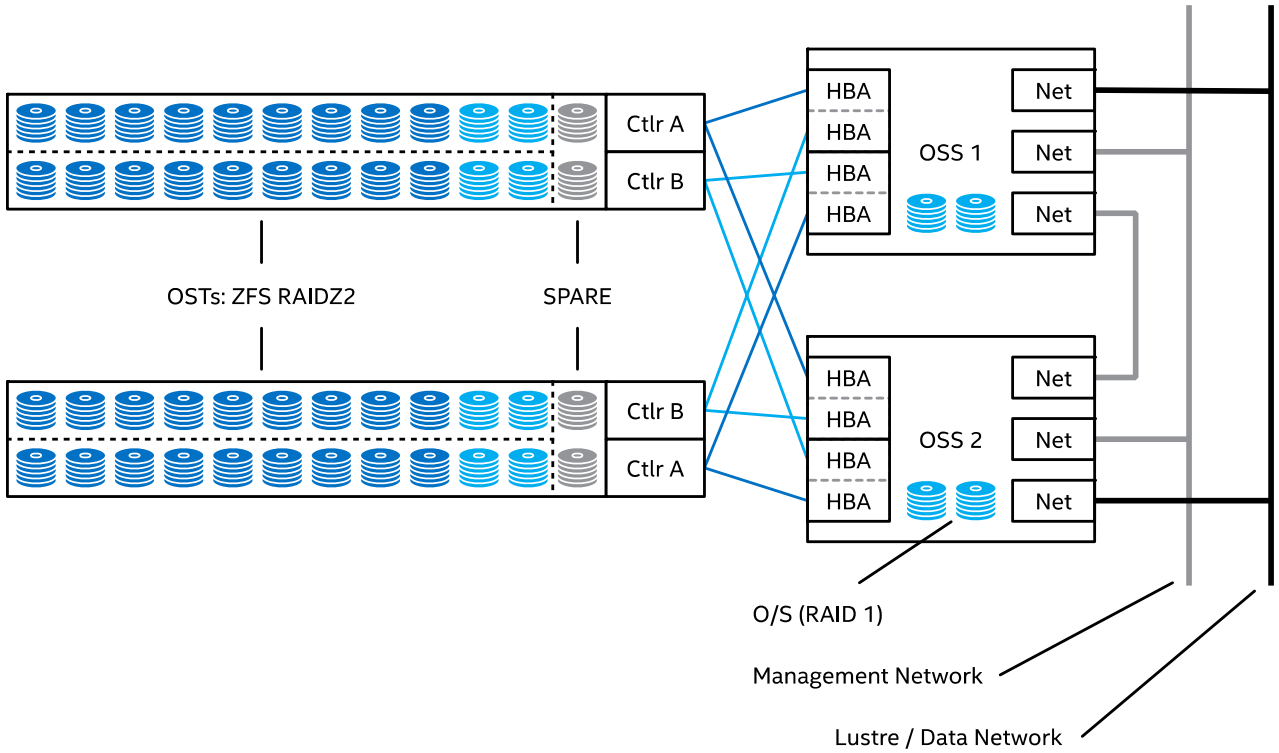

Figure 4. OSS HA Cluster Building Block

Clients

Lustre clients are responsible for creating a coherent, aggregate view of the Lustre file system as a POSIX storage entity. The client establishes connections with the metadata and object storage servers. The metadata for a file includes the file layout information (also referred to as file striping), which is a list of the objects held on the OSTs (each object represents one stripe), and the access pattern. Clients communicate with the OSS servers directly; the MDS does not participate in file IO once the file is opened, until the file is closed.

Lustre uses a cache-coherent distributed lock manager for controlling file IO, ensuring that all Lustre clients can access all files in the file system in parallel, with both read and write concurrency.

Lustre aims for very close compliance with the POSIX standard. From the Lustre Operations Manual:

> *The full POSIX test suite passes in an identical manner to a local EXT4 file system, with limited exceptions on Lustre clients. In a cluster, most operations are atomic so that clients never see stale data or metadata. The Lustre software supports mmap() file I/O.*

Lustre Server Platform Preparation

Overview

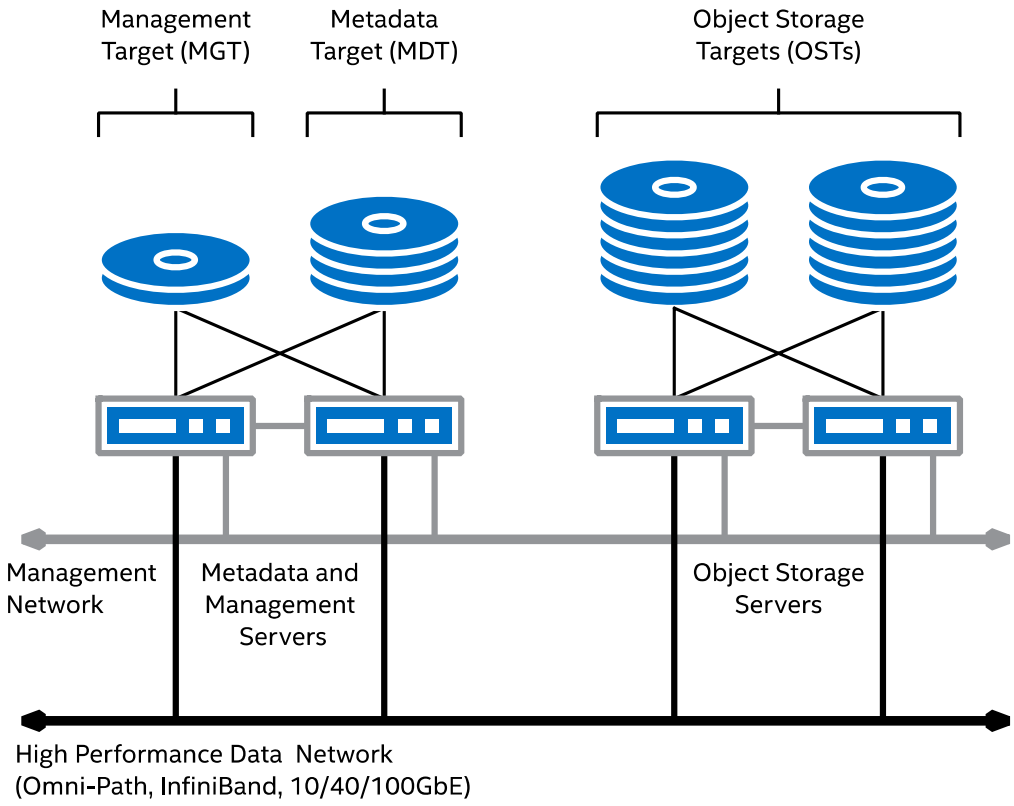

Figure 5. HA Lustre Server Building Blocks

In the reference architecture (depicted in Figure 5) the operating system installation is the same for all servers. Use the Kickstart template in the appendix as a guide for the baseline installation.

Lustre servers commonly have a minimum of two network interfaces:

1. A network for management and maintenance, including software management, health monitoring, remote administrator access.
2. A high-performance data network exclusively for carrying Lustre traffic.

Additional network interfaces can be introduced to provide support for the HA software framework, for example to support additional redundancy in Corosync communications. The hardware and operating system requirements for Pacemaker and Corosync HA framework are described in High Availability and Failover.

The operating system storage requires relatively little capacity, as the footprint of installed packages is quite modest and is unlikely to exceed 10GB overall (this represents something of

an upper limit, and can be revised depending on the final server configuration. Many factors influence the OS payload, including requirements for development tools and additional system management tools). Log file storage must also be accommodated, as well as space for crash dumps, if configured.

A swap partition should also be included in estimates for OS storage needs. Follow the operating system vendor's guide for configuration of swap. In the case of Red Hat, the guidance is to allocate a fixed 4GB swap partition for systems with 64GB or more or system RAM. System performance can degrade markedly when swap is actually used, so the goal is to configure the system with sufficient RAM that it that never needs to use swap. Note that Lustre itself will never use swap, since Lustre is implemented as kernel modules and it is not possible to swap kernel memory. However, there will always be a small number of programs running in user space on the host and because of this, allocating some storage for swap is essential as a contingency.

Two disks, internal to the server chassis and configured as a RAID-1 mirror, provide some fault tolerance. Where possible, use a hardware RAID controller supplied with the server to manage the root disk mirror; this will reduce complexity in the operating system configuration. Otherwise, use LVM to create a root disk mirror. LVM has the added advantage of supporting snapshots, which can be useful when conducting system maintenance, such as a software upgrade. Refer to the documentation from the Linux operating system distribution used for installation for more detailed information on establishing LVM storage volumes.

Lustre servers should be configured with a large amount of system memory in order to take advantage of Lustre's caching features. Metadata servers, in particular, benefit from being able to cache the file system namespace in memory.

While a test system can be configured with as little as 2GB RAM, a production Lustre server should be equipped with at least 64GB for an entry-level platform; ideally 256GB or more should be installed in each server. Insufficient memory capacity can lead to out-of-memory errors when the servers are exposed to demanding, high-performance, production workloads, destabilizing the server and, by extension, the file system.

The block file system type (LDISKFS or ZFS) used by the Lustre servers will also affect memory-sizing considerations. LDISKFS is based on EXT4, and uses a journal device to improve performance. A copy of the journal will also be kept in system memory when the storage is mounted. While the default LDISKFS journal size on OSTs is 400MB, it is common to see OST journals 2-4GB in size due to the performance benefits this brings, particularly for metadata (metadata operations affect OST storage targets to some degree, not just MDTs). The default LDISKFS journal size on the MDT is 4GB. A server must have enough available RAM to hold the journals for all of the storage targets in the HA cluster group, including the storage that will be mounted after a failover event.

For example, if an OSS HA cluster group with two servers has 12 LDISKFS OSTs, each with a 4GB journal, each server will require a minimum of 48GB RAM just for the journals, even

though each server will normally only mount 50% of the OSTs. This is because if one server fails, all the storage targets it normally serves will have to be migrated to the surviving server.

In addition to the journal, cache needs to be reserved, plus metadata space and operating system overheads. To accommodate a journal cache plus metadata on LDISKFS-based servers, a very approximate rule of thumb for the minimum amount of RAM required is: 2.5 times the journal size, times the number of OSTS.

ZFS-based servers also make extensive use of RAM for caching, in particular for the ARC (adaptive replacement cache). By default, up to 50% of the available RAM will be used for the ARC, and this can be tuned as required. Sites have seen good success with as much of 75% of the available RAM allocated for ARC. When configuring ZFS-based Lustre servers, 256GB RAM is recommended.

Network Configuration

Lustre servers are recommended to have 3 network interfaces for high availability configurations, although it is possible to create a Lustre storage server that has only a single network interface. Typically, one network interface is connected to the system management network in order to get access to the broader network infrastructure such as time servers, software package repositories, monitoring and systems management platforms. Lustre traffic is usually contained on a high-speed data network, separate from the management traffic.

A point-to-point or cross-over cable connection between two hosts in a common HA framework is an optional third connection.

Note that Lustre does allow for multi-homed configurations, meaning that servers can be connected to multiple Lustre data networks.

Storage Preparation

In the reference architecture, servers are grouped into pairs, and each pair is connected to a multi-ported storage enclosure. This grouping will be referred to as an HA cluster pair or HA cluster group throughout this documentation. A high availability Lustre file system using the reference architecture in this guide will be comprised of at least two HA cluster groups: one group for the MGS and MDS, usually referred to as the metadata cluster (or individually as metadata servers), and one or more HA groups for the object storage servers.

The storage devices in the enclosure are grouped into Logical Units (LUs), and the logical units are visible to each of the servers in an HA group.

Storage enclosures are usually attached to each of the servers with redundant cable connections, to protect against individual component-level failures, such as a broken cable or host bus adapter (HBA). This is commonly referred to as multipathing. Multipathing configuration is specific to the storage hardware vendor, although some basic guidance is available from the operating system distribution providers. In most Linux distributions,

multipath capability is managed by a software package called Device Mapper Multipath (DM-Multipath). Some storage vendors will provide their own software for managing multipath configuration, but this practice is becoming less and less common.

The device-mapper multipath software is not installed by default on RHEL or CentOS systems using the @core and @base package groups. This software is usually required when working with external storage systems, although one should check with the storage vendor's documentation for information regarding integration of the storage with the Linux distribution in use.

RHEL and CentOS systems provide the multipath software in two packages: `device-mapper-multipath` and `device-mapper-multipath-libs`. To install, use YUM:

```
yum [-y] install device-mapper-multipath
```

YUM will automatically resolve any dependencies and include those packages in the installation manifest. In this case, `device-mapper-multipath` depends on `device-mapper-multipath-libs`, which YUM will automatically include so that the `-libs` package does not have to be specified on the yum command line.

Configuration of multipath devices is out of the scope of this document, but material is available from all of the major Linux distributions, and from several storage hardware manufacturers.

https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html-single/DM_Multipath/

https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html-single/DM_Multipath/

https://access.redhat.com/labsinfo/multipathhelper

Lustre Client Platform Preparation

Overview

Lustre clients have relatively modest configuration requirements in order to connect to a Lustre file system. The number of Linux-based operating systems capable of running the Lustre client software is much broader than that for the Lustre servers, providing more flexibility in the management of compute resources. Lustre has been ported to a variety of Linux distributions and hardware architectures. Nevertheless, the most common deployments use RHEL or CentOS.

The operating system can be installed on a modest internal disk or SSD. Depending on the characteristics of the host's anticipated workload, it may be desirable to make the OS storage fault-tolerant, using a mirrored device, but this is at the discretion of the system owner. Lustre clients can also be diskless because Lustre makes no use of client-side persistence: all I/O is transacted over the LNet protocol between clients and servers. Lustre clients never write directly to storage devices. Because clients can be diskless, the OS footprint is also typically small, at least as far as the Lustre client's requirements are concerned. Lustre clients can use the same base OS template as the servers. An example RHEL kickstart template is included in the appendix.

Lustre client host hardware requirements are generally determined by the operating system distribution and the intended application workload for the host. Lustre can be deployed on a range of platforms, from simple, single-core systems with 2GB RAM, up through multi-socket, multi-core platforms with 512GB+ RAM. The minimum amount of RAM needed for a client is 2GB, but keep in mind when sizing client memory that the application workload and the number of Lustre servers in the file system has an affect on the amount of RAM consumed. For Linux kernel minimum requirements, see the operating system documentation.

Lustre does not make any specific requirements for swap space. Swap allocation guidance should be taken from the OS distribution. Red Hat recommends 4GB for systems with more than 64GB RAM installed. There are, however, examples of HPC installations allocating high-performance flash storage for swap space. Note that for HPC applications, paging out data from memory to swap is generally considered to be a performance impediment.

Network requirements are entirely dependent on the site installation. Some HPC systems limit client connectivity to a single fabric for all communications, including system management, while others separate traffic over two or more network connections, in a manner similar to the Lustre servers. Using multiple fabrics allows the isolation of system-management traffic from application IO.

The reference architecture does not specify networking requirements beyond a single connection to the high-performance data fabric for Lustre communications.

Network Configuration

A single high-speed interface capable of connecting to the Lustre file system network is the minimum requirement for clients. A secondary interface for system management and monitoring (host provisioning, health checking, software updates and job scheduler traffic, etc.) is common but not strictly required.

Operating System Configuration

Overview

This guide does not provide OS management instructions except as they directly relate to the installation and management of Lustre software. Refer to the documentation supplied with the OS for the details of what is required. The guide has been developed using RHEL 7 as the base operating system platform, and all examples have been taken from the same OS unless otherwise stated.

Lustre servers and clients can be configured from a common operating system base. A minimal installation consisting of the @core and @base package clusters is the recommended starting point for both server and client OS installations running RHEL or CentOS.

There is a Kickstart template for the base OS included as an appendix to this guide.

With modern package management systems such as YUM and DNF, package updates and dependency resolution are automatically managed, further simplifying the installation process. It is recommended that the operating system installation be as small and simple as possible, given that additional packages will automatically be installed through dependency resolution when the Lustre packages are installed.

Network Addresses

Lustre servers must have a globally unique and persistent network identifier and this is derived from the IPv4 address of the interfaces used for Lustre network communications. The network interfaces for the Lustre servers must therefore be provided with static IPv4 address allocations. Lustre clients can be assigned static IP addresses or use DHCP. Lustre does not support the use of IPv6 addresses.

Date and Time Synchronization with NTP

While not a strict requirement of Lustre itself, time synchronization across the cluster is very important for overall consistency and coherence. Many applications and file management tools rely on accurate, or at least consistent, time-stamp information. Using NTP to keep time synchronized across the network ensures that time stamps for files are read and written consistently, so that applications get accurate information regardless of where they run in the cluster.

In addition to maintaining consistency in the time stamp records for metadata inodes and file objects, ensuring consistent time representation across a distributed IT infrastructure greatly aids with forensic tasks, such as application debugging or investigations into system failure.

When the hosts all report the same time and date, it is much easier to establish correlations between events reported in the logs for the hosts.

Identity Management

Identity management is an important component of IT infrastructure and cannot be overlooked in Lustre. Users and groups are managed by the host operating system, not by Lustre, and all UIDs and GIDs must be made globally consistent across all Lustre clients and metadata servers. Object storage servers don't have the same requirement, because they do not need to perform permissions checking for Lustre file access.

Any identity services supported by the C library Name Service Switch (NSSwitch) will be compatible with Lustre installations. It is the administrator's choice whether the UNIX identity databases (passwd, shadow, group and gshadow) are used, or a centralized system such as LDAP.

SELinux and Firewall Configuration

For community Lustre versions prior to 2.8, and for Intel® Enterprise Edition for Lustre* software versions older than 3.0.0.0, SELinux is not supported and must be disabled across all servers and clients participating in a Lustre file system. For the Intel® Enterprise Edition for Lustre* software version 3.0.0.0 and later, see the guide: *Configuring SELinux for File Systems based on Intel* EE for Lustre* Software.*

For ease of installation and management, it is suggested that firewall software is disabled. If there is a strong requirement for the operating system firewall to be in place, then make sure that port 988 is open to facilitate LNet communications on TCP/IP infrastructure, and that the NTP port (default: UDP/123) is also open to allow time synchronization.

On Lustre servers using a Pacemaker and Corosync HA framework, ports must be opened to enable Corosync communications and to support the `pcsd` helper daemon for the PCS cluster management software. Instructions on how to do this are provided in the section titled Red Hat Enterprise Linux HA Framework Configuration for Two- Node Cluster. Please refer to the documentation provided by the operating system vendor for further information on the configuration of high availability software on systems where the firewall is enabled.

Firewalls and SELinux add complexity and overheads to installations, and if communications issues appear when setting up an environment, disabling these features as a first step in debugging will often save time in identifying a root cause.

Note that Lustre communications on high-performance fabrics such as Intel® OPA and InfiniBand do not not use TCP/IP for communication, only for node addressing, and are thus not affected by firewall software.

Operating System Software Package Management

Red Hat Enterprise Linux and CentOS both rely heavily on the YUM package manager to install software. Software repositories can be local to the host, in the form of a directory tree or a locally-mounted DVD-ROM or ISO, or made accessible from a network server, usually via the HTTP protocol. Both Red Hat and CentOS maintain repositories accessible via the Internet. CentOS, being a free distribution with no subscription support, provides access to these repositories free of charge. Systems running Red Hat software require an active subscription to the Red Hat Content Delivery Network.

Note that the RHEL High Availability Add-on entitlement is required for Lustre systems that will make use of the Pacemaker and Corosync HA framework software in Red Hat supported systems.

At a minimum, the following subscriptions are required for Lustre systems running RHEL 6:

```
[root@rh6-adm ~]# subscription-manager list
+-------------------------------------------+
    Installed Product Status
+-------------------------------------------+
Product Name:   Red Hat Enterprise Linux High Availability (for RHEL
Server)
Product ID:     83
Version:        6.7
Arch:           x86_64
Status:         Subscribed
Status Details:
Starts:         09/11/15
Ends:           08/11/16

Product Name:   Red Hat Enterprise Linux Server
Product ID:     69
Version:        6.7
Arch:           x86_64
Status:         Subscribed
Status Details:
Starts:         09/11/15
Ends:           08/11/16

[root@rh6-adm ~]# subscription-manager repos --list-enabled
+----------------------------------------------------------+
    Available Repositories in /etc/yum.repos.d/redhat.repo
+----------------------------------------------------------+
Repo ID:   rhel-6-server-rpms
```

```
Repo Name: Red Hat Enterprise Linux 6 Server (RPMs)
Repo URL:
https://cdn.redhat.com/content/dist/rhel/server/6/$releasever/$basea
rch/os
Enabled:   1

Repo ID:   rhel-ha-for-rhel-6-server-rpms


Repo Name: Red Hat Enterprise Linux High Availability (for RHEL 6
Server) (RPMs)
Repo URL:
https://cdn.redhat.com/content/dist/rhel/server/6/$releasever/$basea
rch/highavailability/
           os
Enabled:   1

Repo ID:   rhel-lb-for-rhel-6-server-rpms
Repo Name: Red Hat Enterprise Linux Load Balancer (for RHEL 6
Server) (RPMs)
Repo URL:
https://cdn.redhat.com/content/dist/rhel/server/6/$releasever/$basea
rch/loadbalancer/os
Enabled:   1

Repo ID:   rhel-6-server-optional-rpms
Repo Name: Red Hat Enterprise Linux 6 Server - Optional (RPMs)
Repo URL:
https://cdn.redhat.com/content/dist/rhel/server/6/$releasever/$basea
rch/optional/os
Enabled:   1
```

For RHEL 7-based systems:

```
[root@rh7z-adm log]# subscription-manager list

+-------------------------------------------+
    Installed Product Status
+-------------------------------------------+
Product Name:   Red Hat Enterprise Linux Server
Product ID:     69
Version:        7.2
Arch:           x86_64
Status:         Subscribed
Status Details:
```

```
Starts:         09/11/15
Ends:           08/11/16

[root@rh7z-mds1 ~]# subscription-manager repos --list-enabled
+----------------------------------------------------------+
    Available Repositories in /etc/yum.repos.d/redhat.repo
+----------------------------------------------------------+
Repo ID:   rhel-7-server-rpms
Repo Name: Red Hat Enterprise Linux 7 Server (RPMs)
Repo URL:
https://cdn.redhat.com/content/dist/rhel/server/7/$releasever/$basea
rch/os
Enabled:   1

Repo ID:   rhel-ha-for-rhel-7-server-rpms
Repo Name: Red Hat Enterprise Linux High Availability (for RHEL 7
Server) (RPMs)
Repo URL:
https://cdn.redhat.com/content/dist/rhel/server/7/$releasever/$basea
rch/highavailability/
           os
Enabled:   1
```

To register a subscription entitlement for a server, use the `subscription-manager` command. For example:

```
subscription-manager register --autosubscribe
```

This will automatically select the most suitable subscription for the registered server based on the entitlements granted to the licensee. For more information on managing Red Hat software subscriptions, see the relevant product documentation for the operating system release[5].

The subscription-manager command can also be used to configure specific RHEL package repositories:

```
subscription-manager repos --enable <repo name>
```

[5] RHEL 7: https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/System_Administrators_Guide/
RHEL 6: https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Deployment_Guide/

For example:

```
subscription-manager repos \
  --enable rhel-ha-for-rhel-7-server-rpms
```

Disabling a repository is achieved by using the --disable option in place of --enable:

```
subscription-manager repos --disable <repo name>
```

To get a list of the available RHEL repositories for a given subscription, use the following command:

```
subscription-manager repos --list
```

To get the list of currently enabled repos:

```
subscription-manager repos --list-enabled
```

Using YUM to Manage Software Distribution

To streamline the installation process, the Intel® EE Lustre* package bundles can be copied to an HTTP server on the network and used as local YUM repositories. The bundles distributed with Intel® EE Lustre* software are pre-configured for use as YUM repositories.

Using YUM repositories simplifies the distribution of software packages to computers, aiding provisioning and configuration automation, and simplifying tasks such as auditing and updating.

To install a YUM repository, create a directory for each bundle on a computer that is hosting a web server accessible by the target systems, and extract the tarball into that directory. Apply any configuration changes that may be necessary for the web server to incorporate the new bundle directories. The configuration may need to be reloaded, or the web service restarted when done.

On each server, create a file in `/etc/yum.repos.d` that contains a description for each of the bundles. The configuration file can have any name, provided that it includes the suffix `.repo`. For example, the following entry is for a YUM repository containing Lustre server packages for RHEL 7:

```
[el7-lustre-server]
name = Intel EE Lustre RHEL 7 Lustre Server
baseurl = http://10.70.227.1/el7/lustre-server
enabled = 1
```

There are several options available to describe YUM repositories in the configuration file. Refer to the `yum` and `yum.conf` manual pages for details.

A single file can contain many repository descriptions. Each entry begins with a title contained in square brackets. The title must not contain any white space and should be as simple as possible while still conveying some useful meaning. Shorter titles are preferred, as this helps with output formatting when working with certain YUM commands. There are also times when the operator may wish to temporarily exclude a repository from searches, or only enable certain repositories. Having to type long titles into the command-line to manage the repositories can be cumbersome.

Device Drivers for High Performance Network Fabrics (RDMA, OFED)

Examples herein don't make use of specific 3rd-party device drivers for network interfaces, storage, or other hardware. Where possible, this document references the device driver software supplied by the operating system vendor. There are circumstances where the networking software stack provided by the operating system will need to be replaced by a specific vendor version. This requirement is most common when working with InfiniBand network fabrics, which use specific versions of the OFED software distribution from either the OpenFabrics Alliance or InfiniBand vendors (Intel® or Mellanox*). In this case, the Lustre network drivers need to be recompiled to make use of the 3rd-party network drivers. Building Lustre packages from source is out of the scope of this document but help is available from Intel, or from the Lustre community.

Installing the Lustre Software

Intel® Enterprise Edition for Lustre* software provides packages for servers and clients in compressed tarball bundles for each of the supported OS distributions. The distribution also contains packages for the OpenZFS software developed by the *ZFS on Linux* project[6]. Due to incompatibilities in distribution clauses of the GPL license used by Lustre and the Common Development and Distribution License (CDDL) used by OpenZFS, the ZFS software is distributed as source code. However, the Intel® EE for Lustre* software makes the process of installing ZFS software straightforward.

Because of differences in the installation processes for servers that using the EXT4-based LDISKFS block storage file system format, versus servers using ZFS storage, each is discussed separately in this guide.

---

[6] http://zfsonlinux.org/

Lustre software is also available as source code from the Git repository of the community project, which also provides a set of pre-compiled binary packages. Similarly, the ZFS on Linux project provides source code directly from their project site.

Information on the Lustre community Git repository is here:

https://wiki.hpdd.intel.com/display/PUB/Lustre+Development

The ZFS on Linux project source code is hosted on GitHub:

https://github.com/zfsonlinux

Throughout the instructions in this section of the guide, the Intel® EE for Lustre* software is the Lustre software distribution, and Red Hat Enterprise Linux (RHEL) version 7 is the example host operating system. The same principals of installation and configuration generally apply when working with the community Lustre software and all supported Linux-based operating systems.

Installing Lustre Servers with OpenZFS

OpenZFS support for Lustre Object Storage Devices (OSDs) was introduced in Lustre version 2.4. ZFS is an integrated file system and storage management platform with strong data integrity and volume management features that complement the performance and scalability of Lustre.

The installation process for ZFS-based builds is more complex than for LDISKFS due to complications arising from an incompatibility in the distribution clauses of the licenses for the Linux kernel and OpenZFS. Linux is distributed under the terms of the GPLv2, while OpenZFS is governed by the CDDL. Both GPL and CDDL are free software open source licenses, but certain clauses create an incompatibility that prevents their distribution together in binary form. To accommodate this incompatibility, the ZFS software is therefore distributed as source code.

Fortunately, by making use of a software distribution framework called Dynamic Kernel Modules Support (DKMS), OpenZFS is packaged in a form that is easy for system integrators and operators to build and install. DKMS also ensures that any kernel modules are automatically recompiled if the kernel is updated. Software installation and management for Lustre with ZFS is dependent upon DKMS for successful operation.

Note that for the DKMS mechanism to work, compiler tools and some additional libraries will be needed on each OpenZFS-based Lustre server, regardless of the original build and distribution method. DKMS recompiles DKMS-enabled kernel modules whenever a kernel update is installed, which means the compiler tool-chain must be present on all systems using

the OpenZFS file system. The `kernel-devel` and `kernel-headers` packages for any new Linux kernel are also required.

In the Intel® EE for Lustre* 3.0 software distribution, RHEL 6 and RHEL 7 package bundles are kept in the `el6` and `el7` subdirectories, respectively.

RHEL 6:

```
[root@rh7z-adm ~]# ls -1 ee-3.0.0.0/el6
e2fsprogs-1.42.13.wc4-bundle.tar.gz
iml-agent-3.0.0.0-bundle.tar.gz
iml-manager-3.0.0.0.tar.gz
lustre-2.7.15.3-bundle.tar.gz
lustre-client-2.7.15.3-bundle.tar.gz
robinhood-2.5.5-bundle.tar.gz
zfs-0.6.5.3-bundle.tar.gz
```

RHEL 7:

```
[root@rh7z-adm ~]# ls -1 ee-3.0.0.0/el7
e2fsprogs-1.42.13.wc4-bundle.tar.gz
iml-agent-3.0.0.0-bundle.tar.gz
iml-manager-3.0.0.0.tar.gz
lustre-2.7.15.3-bundle.tar.gz
lustre-client-2.7.15.3-bundle.tar.gz
robinhood-2.5.5-bundle.tar.gz
zfs-0.6.5.3-bundle.tar.gz
```

The following files will be needed from `lustre-<version>-bundle-tar.gz`:

```
lustre-<lu version>-<kernel ver>.el[6,7]_lustre.x86_64.x86_64.rpm
lustre-dkms-<lu version>.el[6,7].noarch.rpm
lustre-osd-zfs-mount-<lu ver>-<kernel>.el[6,7]_lustre.x86_64.x86_64.rpm
lustre-osd-zfs-<lu ver>-<kernel ver>.el[6,7]lustre.x86_64.x86_64.rpm
```

And from the ZFS bundle (`zfs-<version>-bundle.tar.gz`):

```
dkms-<version>.el[6,7].noarch.rpm
spl-<zfs version>.el[6,7].src.rpm
spl-dkms-<zfs version>.el[6,7].noarch.rpm
zfs-<zfs version>.el[6,7].src.rpm
zfs-dkms-<zfs version>.el[6,7].noarch.rpm
```

The OpenZFS bundle contains two source RPM (SRPM) packages, one for ZFS and one for SPL (Solaris Portability Layer). These packages contain source code for ZFS and must be compiled before they can be installed.

The kernel does not require Lustre-specific patches when using ZFS as the storage platform for Lustre servers, so the Lustre-patched kernel is not included in the installer bundle. The ZFS kernel modules will be compiled against the kernel currently running on the target host.

Note: it is strongly recommended that the host operating system always be installed with the latest kernel release supported by the operating system vendor. This ensures that the kernel is protected against known security vulnerabilities and has the latest bug fixes. The Lustre developers work to ensure that Lustre remains compatible across operating system kernel updates for supported platforms.

To compile and install the Open ZFS software for Linux, run the following process on each of the Lustre servers:

1. Install the compiler toolchain and dependencies for ZFS:

   ```
   yum -y install rpm-build zlib-devel libuuid-devel \
   libattr-devel systemd-devel gcc
   ```

   Even if the intention is to create a single build server, separate from the production systems in order to manage package creation, DKMS requires a working development environment on all target hosts, which in this case means all of the Lustre servers that have ZFS OSDs.

2. Install the `kernel`, `kernel-headers`, and `kernel-devel` packages for the target kernel. The process for building the ZFS kernel modules is intended to work with the active, running kernel. Make sure that the target kernel version is installed along with the development packages and that the host has booted using the intended target kernel, otherwise the later steps in the installation process may not be completed correctly and the system will not be left in a consistent state. The target kernel is the version of the operating system kernel that will be used to run the Lustre services and ZFS kernel modules.

   To install the latest version of the OS kernel, use the following command:

   ```
   yum -y install kernel kernel-devel kernel-headers
   ```

   To install a specific version of the kernel, the full version and release number will need to be supplied to the YUM command line for each of the `kernel`, `kernel-devel` and `kernel-headers` packages. For example:

   ```
   yum install \
     kernel-3.10.0-327.13.1.el7 \
   ```

```
  kernel-devel-3.13.0-327.13.1.el7 \
  kernel-headers-3.10.0-327.13.1.el7
```

Reboot the host when the kernel installation is complete.

In following example, the target kernel is the one already active and running on the host, so all that is required is installation of the development RPMs. (In this case, because the target kernel is already running, a reboot is not required):

```
yum install \
  kernel-devel-$(uname -r) \
  kernel-headers-$(uname -r)
```

3. Copy the Lustre server and ZFS bundles from the Intel® EE for Lustre* software distribution onto the host.
4. Untar the `lustre` and `zfs` server bundles into a temporary directory:

```
mkdir -p $HOME/lz
cd $HOME/lz
tar zxf $HOME/lustre-[0-9].*-bundle.tar.gz
tar zxf $HOME/zfs-[0-9].*-bundle.tar.gz
```

5. Build the SPL and ZFS Linux binary packages:

```
cd $HOME/lz
rpmbuild --rebuild spl-[0-9].*.el?.src.rpm
rpmbuild --rebuild zfs-[0-9].*.el?.src.rpm
```

Note: If you examine the spec files for the SPL and ZFS packages, it can be seen that they are only intended to build the user-space tools, not the kernel modules themselves. This is because DKMS will create the kernel modules.

One could build these packages on one machine and then distribute the results to the other hosts, but the time and effort saving is modest at best.

6. Install the DKMS packages and the newly created SPL and ZFS RPMs on the target:

```
cd $HOME/lz
RPMDIR=$(rpm --eval %_rpmdir)
yum -y install \
dkms-*.el?.noarch.rpm \
zfs-dkms-*.noarch.rpm spl-dkms-*.noarch.rpm \
lustre-dkms-*.noarch.rpm lustre-osd-zfs-[0-9]*.rpm \
lustre-osd-zfs-mount-[0-9]*.rpm lustre-[0-9]*.rpm \
```

```
$RPMDIR/x86_64/spl-[0-9].*.el?.x86_64.rpm \
$RPMDIR/x86_64/libnvpair1-[0-9].*.el?.x86_64.rpm \
$RPMDIR/x86_64/libuutil1-[0-9].*.el?.x86_64.rpm \
$RPMDIR/x86_64/libzfs2-[0-9].*.el?.x86_64.rpm \
$RPMDIR/x86_64/libzpool2-[0-9].*.el?.x86_64.rpm \
$RPMDIR/x86_64/zfs-[0-9].*.el?.x86_64.rpm \
$RPMDIR/x86_64/zfs-dracut-[0-9].*.el?.x86_64.rpm
```

Note: The above command line uses regular expressions so that it is portable across version number changes in the packages.

Note: The installation process will take a long time to complete, on the order of several minutes. This is because DKMS compiles the kernel modules. Be sure to factor this time into processes that cover installation of servers, as well as maintenance work and upgrades (updates to the kernel, Lustre or ZFS may trigger a rebuild of the DKMS kernel modules). If a site is subject to the conditions of a service level agreement (SLA) for system availability, this will also need consideration when planning an installation or update.

There is no option on first install through YUM/RPM to create DKMS modules for a specific kernel, only for the kernel currently installed. DKMS effectively requires that the target kernel for the build is installed and running on the host.

One can of course use DKMS to enable matching of any kernel, either through a rebuild or with the weak updates support available on RHEL and CentOS. (The weak updates feature exploits Red Hat's kernel ABI stability guarantee to ensure that kernel symbols remain consistent across kernel updates). However, for the first-time installation, it is far better to build the packages with an exact match to the intended run-time kernel.

7. Verify that the packages have been installed correctly:

   ```
   rpm -qa --last |less
   ```

   The packages should be listed in the `rpm` command output. If not, review the command history to identify what has gone wrong.

8. Verify that DMS is able to recognize the modules and provide status information:

   ```
   dkms status
   ```

   DKMS should report that the `lustre`, `spl` and `zfs` modules are in the "installed" state with a report similar to the following output:

   ```
   [root@rh7z-oss2 ~]# dkms status
   lustre, 2.7.15.3, 3.10.0-327.el7.x86_64, x86_64: installed
   spl, 0.6.5.3, 3.10.0-327.el7.x86_64, x86_64: installed
   ```

```
zfs, 0.6.5.3, 3.10.0-327.el7.x86_64, x86_64: installed
```

If there are multiple kernels installed on the host, there will also be entries listed in the output that see the other kernels. These will have a status of "`installed-weak`" with a reference to the version of the kernel against which the modules were originally compiled. The following example shows a system with two installed kernels:

```
[root@rh7z-oss3 lz]# dkms status
lustre, 2.7.15.3, 3.10.0-327.13.1.el7.x86_64, x86_64: installed
spl, 0.6.5.3, 3.10.0-327.13.1.el7.x86_64, x86_64: installed
zfs, 0.6.5.3, 3.10.0-327.13.1.el7.x86_64, x86_64: installed
lustre, 2.7.15.3, 3.10.0-327.el7.x86_64, x86_64: installed-weak
from 3.10.0-327.13.1.el7.x86_64
spl, 0.6.5.3, 3.10.0-327.el7.x86_64, x86_64: installed-weak from
3.10.0-327.13.1.el7.x86_64
zfs, 0.6.5.3, 3.10.0-327.el7.x86_64, x86_64: installed-weak from
3.10.0-327.13.1.el7.x86_64
```

The entries that have status "`installed`" are modules that have been compiled against the `3.10.0-327.13.1.el7.x86_64` kernel. Entries that have status "`installed-weak`" are exploiting the weak-updates support in the OS kernels that enables the modules to be loaded into a kernel that is different from the one for which the module was originally compiled.

9. Review the module information for one of the Lustre modules and one of the ZFS modules. (Note that the following output may differ slightly from the output observed on the target system and is provided as an example only.):

```
modinfo lustre
modinfo zfs
For example:
[root@rh7z-oss3 ~]# modinfo lustre
filename:       /lib/modules/3.10.0-
327.13.1.el7.x86_64/extra/lustre.ko
license:        GPL
description:    Lustre Lite Client File System
author:         Sun Microsystems, Inc. <http://www.lustre.org/>
rhelversion:    7.2
srcversion:     ED55C746EC393D287A01DAA
depends:        obdclass,ptlrpc,libcfs,lov,lmv,mdc,lnet
vermagic:       3.10.0-327.13.1.el7.x86_64 SMP mod_unload
modversions

[root@rh7z-oss3 ~]# modinfo zfs
```

```
filename:       /lib/modules/3.10.0-
327.13.1.el7.x86_64/extra/zfs.ko
version:        0.6.5.3-1
license:        CDDL
author:         OpenZFS on Linux
description:    ZFS
rhelversion:    7.2
srcversion:     CEB8F91B3D53F4A2844D531
depends:        spl,znvpair,zcommon,zunicode,zavl
vermagic:       3.10.0-327.13.1.el7.x86_64 SMP mod_unload
modversions
…
<output truncated for brevity>
```

If the installation fails, check the following:

1. If the output of `dkms status` shows components as added, but not installed, then try to use the `dkms` command to manually install the affected packages. The following example shows that none of the DKMS modules are properly integrated into the operating system, although each module package has been added to the host:

   ```
   [root@rh7z-oss2 ~]# dkms status
   lustre, 2.7.15.3: added
   spl, 0.6.5.3: added
   zfs, 0.6.5.3: added
   ```

2. To attempt to install the modules into the kernel, use the `dkms install` command, starting with the `spl` module, then `zfs`, and the `lustre` module last of all. If some modules are already marked as installed, they can be skipped. To install a module into the running kernel, use the following syntax:

   ```
   dkms install <module name>/<module version>
   ```

   For example:

   ```
   dkms install spl/0.6.5.3
   ```

3. If the manual install step does not work, it may be necessary to build the modules again first. For the most consistent results, run `dkms build` for each of the modules in order: `spl`, then `zfs`, then `lustre`. The syntax is as follows:

   ```
   dkms build <module name>/<module version>
   ```

   For example:

```
dkms build spl/0.6.5.3
```

Note that after the module is built, it will need to be installed using the `dkms install` command. The `dkms build` command is not often used directly, because the `install` command will also attempt to build the DKMS modules if they have not been already.

4. The Lustre DKMS module is the most complex due to its dependencies on both the SPL and ZFS modules, as well as the OS kernel and in some cases 3rd party kernel modules for storage and network devices. It is, therefore, the most susceptible to issues when the build environment is not correctly configured.

   If the `lustre-dkms` package did not install correctly and the Lustre DKMS module is not in the "installed" state, it may have a corrupted DKMS configuration. The most typical symptom of a corrupt Lustre DKMS configuration occurs when execution of either the `dkms build` or `dkms install` command on the Lustre module fails with output similar to the following:

   ```
   dkms.conf: Error! Directive 'DEST_MODULE_LOCATION' does not begin with
   /kernel', '/updates', or '/extra' in record #0.
   ```

   There may be several similar lines of output. In this case, the only safe action is to remove all of the Lustre RPMs, verify that the run-time environment meets the minimum requirements for Lustre and ZFS, and then re-install the Lustre RPMs. To remove the Lustre packages, run:

   ```
   rpm -e lustre lustre-dkms lustre-osd-zfs lustre-osd-zfs-mount
   ```

   Check that the SPL and ZFS packages are correctly installed and configured:

   ```
   dkms status
   ```

   Healthy status for the spl and zfs modules will look similar to this output:

   ```
   [root@rh7z-oss2 ~]# dkms status
   spl, 0.6.5.3, 3.10.0-327.el7.x86_64, x86_64: installed
   zfs, 0.6.5.3, 3.10.0-327.el7.x86_64, x86_64: installed
   ```

   Re-install the Lustre RPMs:

   ```
   yum install \
   lustre-[0-9].*.el?_lustre.x86_64.x86_64.rpm \
   lustre-dkms-[0-9].*.el?.noarch.rpm \
   lustre-osd-zfs-[0-9].*.el?_lustre.x86_64.x86_64.rpm \
   lustre-osd-zfs-mount-[0-9].*.el?_lustre.x86_64.x86_64.rpm
   ```

> As mentioned before in the main installation process, the installation of the Lustre packages can take a long time to complete, as the DKMS kernel modules will be compiled from source code.

The Intel® EE for Lustre* software distribution includes helper scripts to assist with the installation process for ZFS installations. There are two basic scripts that are relevant. First is a script that will create the basic software bundle that is distributed to each server.

To create the distribution bundle, extract the main Intel® EE for Lustre* software tarball, then run the script `create_installer`, which is inside the top level directory of the extracted tarball. The following example shows how to create a ZFS server distribution bundle for Intel® EE for Lustre* software, version 3.0. Enter the command appropriate for your software version.

```
[root@rh7z-mds1 ~]# tar zxf ee-3.0.0.0.tar.gz
[root@rh7z-mds1 ~]# cd ee-3.0.0.0
[root@rh7z-mds1 ee-3.0.0.0]# ./create_installer zfs
```

The `create_installer` script will generate gzip-compressed tarballs for distribution to the Lustre storage servers, one for each of RHEL 6 and RHEL 7:

```
lustre-zfs-el6-installer.tar.gz
lustre-zfs-el7-installer.tar.gz
```

The script repackages the Lustre RPMs with the OpenZFS RPMS and creates a new bundle. This can be distributed to the hosts that are to be configured as Lustre servers.

The second script, called `install`, is part of the bundle generated by `create_installer` and encapsulates the requisite `yum` and `rpmbuild` commands to create binary packages and installs them on the target host.

The Intel® EE for Lustre scripts will not update the kernel on target hosts, so make sure that all of the Lustre server hosts are loaded with and running the latest kernel release distributed by the OS vendor or the target version mandated by the system design. (The OS release and package versions may be subject to strict compliance and audit controls for a given site in an organization, or there may be a project or program restriction that governs the operating system software.)

After it's created, distribute the bundle to all of the target servers and run the install script on each target. The following example creates the bundle on an admin server, copies the result over to an OSS, and runs the install script (output is truncated for brevity).

Create the Lustre ZFS server bundle and copy it to the target OSS node:

```
[root@rh7z-adm ~]# cd ee-3.0.0.0/
```

```
[root@rh7z-adm ee-3.0.0.0]# ./create_installer zfs
[root@rh7z-adm ee-3.0.0.0]# scp lustre-zfs-el7-installer.tar.gz rh7z-
oss4:
lustre-zfs-el7-installer.tar.gz                              100%   13MB
13.3MB/s   00:00
```

On the target node:

```
[root@rh7z-oss4 ~]# yum -y install kernel kernel-devel kernel-
headers >/dev/null 2>&1
reboot
```

After reboot, login and execute the installer:

```
[root@rh7z-oss4 ~]# tar zxf lustre-zfs-el7-installer.tar.gz
[root@rh7z-oss4 ~]# cd lustre-zfs/
[root@rh7z-oss4 lustre-zfs]# ls
dkms-2.2.0.3-28.git.7c3e7c5.el6.noarch.rpm
install
lustre-2.7.15.3-3.10.0_327.13.1.el7_lustre.x86_64.x86_64.rpm
lustre-dkms-2.7.15.3-1.el6.noarch.rpm
lustre-osd-zfs-2.7.15.3-3.10.0_327.13.1.el7_lustre.x86_64.x86_64.rpm
lustre-osd-zfs-mount-2.7.15.3-
3.10.0_327.13.1.el7_lustre.x86_64.x86_64.rpm
spl-0.6.5.3-1.el7.src.rpm
spl-dkms-0.6.5.3-1.el7.noarch.rpm
zfs-0.6.5.3-1.el7.src.rpm
zfs-dkms-0.6.5.3-1.el7.noarch.rpm
[root@rh7z-oss4 lustre-zfs]# ./install
```

When the installation is finished, the server is ready to run Lustre with ZFS storage:

```
[root@rh7z-oss4 lustre-zfs]# dkms status
lustre, 2.7.15.3, 3.10.0-327.13.1.el7.x86_64, x86_64: installed
spl, 0.6.5.3, 3.10.0-327.13.1.el7.x86_64, x86_64: installed
zfs, 0.6.5.3, 3.10.0-327.13.1.el7.x86_64, x86_64: installed
lustre, 2.7.15.3, 3.10.0-327.el7.x86_64, x86_64: installed-weak from
3.10.0-327.13.1.el7.x86_64
spl, 0.6.5.3, 3.10.0-327.el7.x86_64, x86_64: installed-weak from
3.10.0-327.13.1.el7.x86_64
zfs, 0.6.5.3, 3.10.0-327.el7.x86_64, x86_64: installed-weak from
3.10.0-327.13.1.el7.x86_64
[root@rh7z-oss4 lustre-zfs]# modinfo lustre
```

```
filename:       /lib/modules/3.10.0-
327.13.1.el7.x86_64/extra/lustre.ko
license:        GPL
description:    Lustre Lite Client File System
author:         Sun Microsystems, Inc. <http://www.lustre.org/>
rhelversion:    7.2
srcversion:     ED55C746EC393D287A01DAA
depends:        obdclass,ptlrpc,libcfs,lov,lmv,mdc,lnet
vermagic:       3.10.0-327.13.1.el7.x86_64 SMP mod_unload
modversions
[root@rh7z-oss4 lustre-zfs]# modinfo zfs
filename:       /lib/modules/3.10.0-327.13.1.el7.x86_64/extra/zfs.ko
version:        0.6.5.3-1
license:        CDDL
author:         OpenZFS on Linux
description:    ZFS
rhelversion:    7.2
srcversion:     CEB8F91B3D53F4A2844D531
depends:        spl,znvpair,zcommon,zunicode,zavl
vermagic:       3.10.0-327.13.1.el7.x86_64 SMP mod_unload
modversions
…
```

Lustre Client Software Installation

The Lustre client software comprises a package containing the kernel modules and a package of user-space tools used to manage the client software. In the Intel® EE for Lustre* software, these packages are collected into client bundles, one bundle for each supported operating system.

In the Intel® EE for Lustre* 3.0 distribution, RHEL 6 and RHEL 7 package bundles are kept in the `el6` and `el7` subdirectories, respectively.

RHEL 6:

```
[root@rh7z-adm ~]# ls -1 ee-3.0.0.0/el6/lustre-client*
lustre-client-2.7.15.3-bundle.tar.gz
```

RHEL 7:

```
[root@rh7z-adm ~]# ls -1 ee-3.0.0.0/el7/lustre-client*
lustre-client-2.7.15.3-bundle.tar.gz
```

The Lustre client kernel module packages are compiled against specific kernel versions but because the kernel for Lustre clients does not require Lustre-specific patches, there is no Linux kernel included in the client bundles. Instead, when the Lustre client kernel modules are installed using package management software (YUM on RHEL-based systems), the appropriate kernel will be automatically included in the installation manifest as part of the dependency resolution step. For this to work, the operating system must be configured to install packages using package repositories, usually via the network. If this is not possible, then the dependencies will need to be downloaded and installed manually.

Note: as for all Lustre assets, it is strongly recommended that the host operating system always be installed with the latest kernel release supported by the operating system vendor. This ensures that the kernel is protected against known security vulnerabilities and has the latest bug fixes. The Lustre developers work to ensure that Lustre remains compatible across operating system kernel updates for supported platforms.

All of the Lustre clients should be installed using the same procedure. To install the Lustre client software, run the following process on each of the Lustre client hosts:

1. Extract the Lustre distribution tarball into a temporary directory on one of the servers. For example:

   ```
   tar zxf ee-3.0.0.tar.gz
   ```

2. Copy the `lustre-client` package bundle for the target OS onto each of the Lustre clients. For example, to copy the client bundle named `el7/lustre-client-2.7.15.3-bundle.tar.gz` to four hosts named `rh7z-c1`, `rh7z-c2`, `rh7z-c3`, `rh7z-c4`:

   ```
   cd ee-3.0.0
   for i in {1..4}; do
     scp el7/lustre-client-2.7.*-bundle.tar.gz root@rh7z-c$i:
   done
   ```

3. On each client host, execute the following commands to install the Lustre client packages. The installation command must be run with super-user privileges:

   ```
   mkdir -p $HOME/lu
   cd $HOME/lu
   tar zxf $HOME/lustre-client-2.7.*-bundle.tar.gz
   yum [-y] localinstall lustre-client-?.* lustre-client-modules-*
   ```

4. Verify that the installation of the packages has completed successfully by reviewing the package installation history on the host:

   ```
   rpm -qa --last | less
   ```

This will show the list of packages installed in descending date order, newest entry at the top. Installing the `lustre-client` and `lustre-client-modules` packages will also trigger the installation of the matching `kernel` package and some additional tools and libraries, notably `libyaml`, `net-snmp-libs` and `net-snmp-agent-libs`.

5. When installation of the software is complete, and if the operating system kernel was updated, reboot the host so that the OS boots from the newly-installed kernel.
6. When the system returns to operation after a reboot, the running kernel should now be the version that was installed during this process. If this is not the case, then review the RPM database, the YUM installation log (`/var/log/yum.log`) and the syslog (`/var/log/messages`) to ensure that the packages were installed without error. Also check the GRUB boot loader configuration to ensure that there is a valid entry for the new kernel included in the configuration, and that the new kernel is set as the default.

If the required kernel version (matching the kernel against which the Lustre client modules were compiled) is not available, it may have been archived to a maintenance version of the operating system (for example, when minor releases of CentOS are made, the previous versions are archived to repositories at http://vault.centos.org).

RHEL has an equivalent set of repositories for older OS releases that are not made available as part of the normal subscription entitlements.

Generally, if the kernel version is not found by YUM, it indicates that the packages being considered for installation are from an older release of operating system. Because kernels are routinely patched for security and other bug fixes, users should always attempt to install the latest versions of the kernel packages and Lustre client packages. If a Lustre build is unavailable for the latest kernel release, it can be requested through a Lustre integration partner or by rebuilding the packages from source. A source code package is included in the Intel® EE Lustre* client distribution for this purpose.

Configure Lustre Networking (LNet)

Introduction to Lustre Networks

Lustre's network communication protocol, LNet, was originally derived from a project called Portals, developed by Sandia National Labs. LNet is designed to be lightweight and efficient and supports message passing for RPC request processing and RDMA for bulk data movement. All network metadata and file data I/O is managed through the LNet protocol and API. LNet is lightweight and versatile, capable of operating over different network fabrics, including Ethernet, InfiniBand and Intel® OPA. In common with the other major components of Lustre, LNet is implemented as a Linux kernel module.

All participants in a Lustre file system, including servers and clients, must have a valid LNet configuration and be connected either directly on a common network fabric, or via a router between networks.

To support the various types of networks, LNet has a low-level device layer called a Lustre Network Driver (LND), implemented as a pluggable driver module. The LND provides an interface abstraction between the upper level LNet protocol and the kernel device driver for the network interface. Each low-level network protocol requires a separate LND and multiple LNDs can be active on a host simultaneously, if the server requires access to more than one type of network. The most commonly used LNet drivers are the `ksocklnd.ko` module for TCP/IP networks, and the `ko2iblnd.ko` module for RDMA networks that make use of the OpenFabrics Enterprise Distribution (OFED) network driver. The `ksocklnd.ko` module is usually abbreviated to `socklnd` or referred to as the sockets LND. The `ko2iblnd.ko` module (usually referred to as the `o2ib` LND or just called `o2ib`) supports fabrics running InfiniBand, Omni-Path, and RDMA over Converged Ethernet (RoCE). The letter "k" prefix in the LND names is there to emphasize that these are kernel modules, which is true for the majority of the Lustre software stack. It is often omitted from documentation, to improve readability.

LNet also supports the ability to route Lustre communications between different networks. Dedicated computers called LNet Routers (once again, imagination in the naming of services makes way for ruthless pragmatism) can direct traffic between multiple LNets.

Network interfaces on computer systems running LNet are addressed with a node identifier (the IPv4 address of a network device on the host) and also with a protocol identifier and network number for that protocol. The format is as follows:

```
<IPv4 address>@<LND protocol><lnd#>
```

The complete string is called an LNet Network Identifier (NID) and it uniquely defines an interface for a host on an LNet communications fabric.

The following example is a NID for an Ethernet interface:

```
10.70.207.11@tcp0
```

This is a unique ID for a host with an Ethernet NIC on LNet `tcp0` using the socket LND (as indicated by the `tcp` network type). A second configuration could also be added for a different interface:

```
192.168.70.11@tcp1
```

The number appended to the LND protocol type must be a non-negative integer and must be the same for all Lustre hosts (client and server) that participate on the same network. For example, the following two NIDs are *not* on the same LNet:

```
10.70.207.11@tcp0
10.70.207.11@tcp1
```

Even though each NID has the same IPv4 address, they belong to two different LNets, because the LND instance numbers are different (`tcp0` and `tcp1`, respectively).

The inverse case is not true: it is not possible for multiple interfaces to belong to the same LNet on a host. For example, the following NIDs cannot be created on the same host:

```
10.70.207.11@tcp0
10.70.207.22@tcp0
```

If there is a requirement for a host to have multiple interfaces connected to the same subnet, then network bonding of the underlying devices should be used, with the resulting bonded interface then being presented to the LNet configuration. Note that for InfiniBand fabrics, the kernel bonding driver only supports active-passive, or failover, operation. That is, only one interface in the bond can be used to send or receive traffic at a given point in time.

RDMA networks have a similar NID format. For example:

```
10.20.30.11@o2ib0
```

Note that even though RDMA is being used for communications in the o2ib driver, the LNet NID still makes use of an IPv4 address to identify the NID. In fact, LNet requires that the HCA or HFI is configured with an IPv4 address, even though the LND does not use the IPoIB upper level protocol (ULP). LNet uses the IP address to identify which physical interface to use for communications – it is a convenient label for the device. Arguably, LNet could have implemented the NID using the MAC address or GUID of the device, but using an IPv4 address provides a level of consistency in defining the interfaces, regardless of the underlying network fabric or protocol.

LNet Configuration Overview

LNet configuration is straightforward, and it is easy to create a working LNet environment. After installing the Lustre packages, configuration of LNet is the first task an administrator will perform on a host.

In releases of Lustre prior to version 2.7.0, configuration of the LNet device driver is managed exclusively by supplying options to the `lnet` kernel module. The options are read from a `modprobe` configuration file when the kernel module is first loaded. With the release of Lustre 2.7.0, administrators can instead make use of the Dynamic LNet feature, controlled by a utility called `lnetctl`, as an alternative to static modprobe options files. Both methods are covered in the following sections.

If no explicit configuration is supplied to LNet, it will attempt to create a valid TCP/IP (socklnd) NID for LNet tcp0 using the first network interface that is detected by the operating system (e.g. `eth0`) when the module is loaded and the LNet service started. The order of interface detection is entirely at the discretion of the operating system, which means that there is no guarantee that the ordering of interfaces will be preserved between reboots and on the insertion of a new hardware device. It also means that the default behavior for a host will differ depending on its hardware configuration. Most operating systems do try to ensure that a device, after it is detected, maintains the same device name (eth0, eth1, etc.) between reboots. Nevertheless, it is strongly recommended that all configuration be stated explicitly: defining the configuration also defines the expected behavior of the system, making it easier to audit.

Configuration of LNet Using Modprobe Options Files

To configure an LNet interface by supplying parameters directly to the `lnet` module when it is loaded by the kernel, create a file in the normal `modprobe` configuration options directory (usually `/etc/modprobe.d`), and add an entry for the LNet module. By convention, Lustre module configuration options are recorded in a file called `/etc/modprobe.d/lustre.conf` (although tuning parameters for devices can be separated into device-specific files, such as `/etc/modprobe.d/ko2iblnd.conf` – see the section Optimizing o2iblnd Performance for an example of this, which shows an example of an optimized configuration that is automatically applied when an Intel® Omni-Path interface is detected).

There are two ways to define an LNet configuration using module options: `networks` and `ip2nets`. The `networks` parameter is the easiest to understand and uses a very simple syntax, while `ip2nets` gives administrators the greatest flexibility at the expense of much greater complexity. The `networks` syntax is strongly recommended unless there is a specific requirement that can be more readily fulfilled by the `ip2nets` syntax.

There should be only one `networks` or `ip2nets` parameter defined for the `lnet` module. The parameters are mutually exclusive; choose one or the other but not both. If both options

are referenced in the modules configuration, then the following warning will be reported in the kernel ring buffer (dmesg) and system console:

```
[46290.641911] LNetError: 101-0: Please specify EITHER 'networks' or
'ip2nets' but not both at once
[46290.645288] LNetError:
3483:0:(config.c:199:lnet_parse_networks()) networks string is
undefined
```

The second line in the error output is a side-effect of the conflict between `ip2nets` and `networks` options. The key error report is the first line of the above output. Remove one or other of the `networks` or `ip2nets` options to resolve the conflict.

If there is more than one `networks` or `ip2nets` declaration defined through successive entries in one or more kernel module configuration files (i.e., multiple `networks` declarations or multiple `ip2nets` declarations), the last entry that is read by the module loader will be used. Make sure that there is no more than one LNet configuration defined in the modprobe directory (`/etc/modprobe.d` in RHEL and CentOS) when using this method to define LNet parameters.

Kernel module options are read only when the kernel module is loaded, passed to the kernel by the module loader. As a consequence, changes in the configuration will not be applied until the `lnet` module is stopped, unloaded from the kernel, and then reloaded.

LNet networks syntax

The simplest configuration has the following syntax:

```
options lnet networks="<lnd><#>(<dev>)[,…]"
```

For example, to add a TCP/IP (`socklnd`) LNet NID for an Ethernet device using the `eth1` NIC:

```
options lnet networks="tcp0(eth1)"
```

Notice that the IP address is not mentioned in the configuration, only the device name. The syntax is simple and readable.

The next example creates an RDMA verbs (`o2iblnd`) LNet interface:

```
options lnet networks="o2ib0(ib0)"
```

If there is more than one network interface on the host that will carry LNet traffic, additional interfaces can be added to the `networks` variable:

```
options lnet networks="tcp0(eth1),o2ib0(ib0),o2ib1(ib1)"
```

The above example configures three LNet interfaces: a `socklnd` NID on `eth1`, and `o2iblnd` NIDs for `ib0` and `ib1`.

To create a configuration where one physical interface belongs to two LNets, specify each LNet in a comma-separated format, with the same physical interface in brackets for each LNet. For example:

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

This is not a common configuration, because its practical application is limited (both LNets will be on the same subnet and connect to the same physical interface, so there is no advantage in creating two LNets in this case).

LNet ip2nets syntax

The `ip2nets` configuration syntax makes use of regular expressions to programmatically create a host's LNet configuration from a single global definition. The administrator creates pattern-matching rules based on the IPv4 address of the network interfaces of the target hosts, and the first matching rule is used to determine the configuration that will be applied. In this way, a single configuration file can be applied to both clients and servers across a heterogeneous network environment, and across multiple networks. The IP addresses are only used to identify the network interfaces on the target hosts, and are not used for communications purposes. All Lustre communications are managed by the LNet protocol, regardless of the underlying transport.

This flexibility comes at the cost of simplicity and readability. The `ip2nets` rules can be complex, making it difficult to interpret the effects of a rule on a given set of hosts. A syntax error introduced into the rules, such as an incorrectly scoped pattern matching expression, can have a wide-ranging, negative impact on the configuration of the entire host population, if it is distributed before the error is caught. Testing the `ip2nets` configuration is more complex as a result.

The general syntax is as follows:

```
options lnet ip2nets="<lnd>[<#>][(<dev>)][, <lnd>[<#>][(<dev>)]]
<pattern>[; …]"
```

Each rule is separated by a semi-colon and rules are evaluated in left-to-right order when `lnet` is started. The first match for each LNet will be applied to the configuration. Evaluation continues until all unique LNets are created or the last rule has been evaluated. This allows a single rule to define multiple NIDs for a single server, each one for a different LNet on a different network interface.

It is simplest to illustrate how to use ip2nets with an example. The following describes a configuration for two different LNets on different types of network fabric:

```
options lnet ip2nets="tcp0(eth1) 10.10.100.*; o2ib0(ib0)
192.168.200.*"
```

Hosts having an IP address that matches the pattern 10.10.100.* will be configured with a NID on the tcp0 LNet, and hosts that match the 192.168.200.* pattern will be configured with an o2ib0 LNet. Should a host match both patterns, then both LNets will be configured for that host.

The pattern-matching syntax includes numeric ranges in the format [x-y], for example [2-20]. This can be further refined with a divisor to allow for stepping in the range: [2-20/2] will match all of the even numbers in the range, and [1-19/2] can be used to match all of the odd numbers in a range.

In the next example, some of the hosts will use eth0 while others use eth1 to connect to the LNet tcp0. This configuration is most likely when there are hardware differences between systems and they are using different NICs to connect to the same subnet:

```
options lnet ip2nets="tcp0(eth0) 10.10.100.[1-50]; tcp0(eth1)
10.10.100.[100-200]"
```

If the interface definition is omitted, then the host interface that matches the IP address will be used. So the above example could be re-written as:

```
options lnet ip2nets="tcp0 10.10.100.*"
```

If no match is found in the configuration, LNet will report an error:

```
[257641.245272] LNetError: 11a-a: ip2nets does not match any local
IP interfaces
[257641.247813] LNetError:
8881:0:(config.c:199:lnet_parse_networks()) networks string is
undefined
```

Contrary perhaps to expectations, if `ip2nets` does not find a match, the LNet kernel module will not fall back to the networks option or load a default configuration. A failure to match a pattern in the `ip2nets` options declaration is considered to be implicit confirmation that the target system is not meant to have an LNet configuration applied.

A more complex demonstration:

```
options lnet ip2nets="tcp0(eth1) 10.70.207.[11,12]; \
    tcp0(eth2) 10.70.207.[21-24]; \
    tcp0 10.70.*.*"
```

There are two hosts that will create an LNet NID on the `eth1` device, four hosts using `eth2`, and the remainder of the `10.70/16` network will create an LNet NID for the `tcp0` LNet on the first interface presented by the operating system, provided that the interface has an IPv4 address on the `10.70/16` subnet.

This last item requires further explanation. The configuration entry "tcp0 10.70.*.*" is open to misinterpretation, as the syntax is ambiguous to a human reader. Contrary to expectation, this does not mean "configure the interface that matches the pattern `10.70.*.*` as LNet `tcp0`". It is in fact far more specific. It means "configure the first interface detected by the host operating system as LNet `tcp0` if it also matches the pattern `10.70.*.*`". If either condition is false, then no LNet NID will be configured.

One can also mix LNDs in the ip2nets syntax, should this be required:

```
options lnet ip2nets="o2ib0(ib0) 192.168.207.[11,12]; \
    o2ib0(ib2) 192.168.207.[21-24]; \
    tcp0 10.70.*.*"
```

Comments are also supported, but take care with placement of the comment relative to the semi-colon that separates rules. A comment marker instructs the parser to ignore everything between the comment marker and the next semi-colon in the string, or the end of the string, whichever comes first. If a comment is started after a semi-colon, that could have the effect of causing the parser to ignore an important part of the configuration.

For example, the following syntax is correct:

```
options lnet ip2nets="tcp0(eth1) 10.70.207.[11-12] # MGS/MDS;
\
    tcp0(eth2) 10.70.207.[21-24] # OSS; \
    tcp0 10.70.*.* # Everything else"
```

In the next example, the second line will be ignored because the comment marker (#) is placed after the semi-colon, causing the next line to be treated as comment text, not configuration syntax:

```
options lnet ip2nets="tcp0(eth1) 10.70.207.[11-12] ; # MGS/MDS \
    tcp0(eth2) 10.70.207.[21-24] # OSS; \
    tcp0 10.70.*.* # Everything else"
```

All of the text highlighted in the example will be treated as a comment

Starting and Stopping LNet

LNet, like most of the services that comprise a Lustre file system, runs in the Linux kernel and is incorporated as a kernel module. LNet is started in two steps:

1. Load the kernel modules
2. Start the services

The `lnet` kernel module can be loaded directly through the modprobe command or indirectly by loading a kernel module that has a dependency on LNet. In normal operation, the `lnet` module will be loaded indirectly as a consequence of attempting to start a Lustre service, e.g. by mounting a file system on a client. However, one can treat LNet as independent of Lustre and start it on its own. This is useful for testing and debugging purposes, and to provide some verification of correctness when a system boots up prior to committing to loading the higher-level services (i.e. Lustre).

To load the LNet kernel module, run:

```
modprobe [-v] lnet
```

The `-v` flag is optional and provides verbose output. This is useful for debugging purposes, but it is normally omitted. For example:

```
[root@rh7z-pe ~]# modprobe -v lnet
insmod /lib/modules/3.10.0-
327.13.1.el7_lustre.x86_64/extra/kernel/net/lustre/libcfs.ko
insmod /lib/modules/3.10.0-
327.13.1.el7_lustre.x86_64/extra/kernel/net/lustre/lnet.ko
networks="tcp0(eth1)"
```

Notice that a second module, called `libcfs.ko`, was also loaded. The `libcfs` module is an API used throughout Lustre and LNet and provides primitives for things like process management, memory management, and debugging.

After the module is loaded, the LNet service needs to be started:

```
lctl network up
# or
lctl network configure
```

The `lctl network` command works on all versions of Lustre, and prior to version 2.7, it is the only way to manually start LNet. In Lustre 2.7 and onward, there is also the `lnetctl` utility:

```
lnetctl lnet configure [--all]
```

The `lnetctl configure` command will not automatically configure networks that are specified in the kernel module parameters; the `lnet` service will start, but the interfaces will not be configured. Supplying the `--all` flag will cause all of the networks defined as kernel module options to be loaded and started.

To view the loaded configuration:

```
lctl list_nids

# or for dynamic lnet in Lustre 2.7+
lnetctl net show [--verbose]
lnetctl export
```

The `lnetctl export` command is equivalent to `lnetctl net show --verbose`.

To shut down LNet and unload the kernel modules, first stop the LNet networks on the host:

```
lctl network down
# or
lctl network unconfigure
```

Then use the `lustre_rmmod` command to unload the kernel modules:

```
lustre_rmmod
```

One can unload the module by using `rmmod` directly:

```
rmmod lnet
rmmod libcfs
```

The `lustre_rmmod` is the recommended method for unloading Lustre and LNet kernel modules, because it will check for dependencies and eliminates any guesswork on the part of the systems administrator in trying to identify all of the modules to unload and the correct sequence for doing so.

LNet can also be loaded indirectly, as a dependency of the `lustre` kernel module. If LNet is loaded in this way, its start-up behavior is different, as the LNet networks defined in kernel module options will be automatically configured and brought online. This is easily illustrated just by loading the Lustre module:

```
[root@rh7z-pe ~]# modprobe -v lustre
insmod /lib/modules/3.10.0-
327.13.1.el7_lustre.x86_64/extra/kernel/net/lustre/libcfs.ko
```

```
insmod /lib/modules/3.10.0-
327.13.1.el7_lustre.x86_64/extra/kernel/net/lustre/lnet.ko
networks="tcp0(eth1)"
insmod /lib/modules/3.10.0-
327.13.1.el7_lustre.x86_64/extra/kernel/fs/lustre/obdclass.ko
insmod /lib/modules/3.10.0-
327.13.1.el7_lustre.x86_64/extra/kernel/fs/lustre/ptlrpc.ko
insmod /lib/modules/3.10.0-
327.13.1.el7_lustre.x86_64/extra/kernel/fs/lustre/fld.ko
insmod /lib/modules/3.10.0-
327.13.1.el7_lustre.x86_64/extra/kernel/fs/lustre/fid.ko
insmod /lib/modules/3.10.0-
327.13.1.el7_lustre.x86_64/extra/kernel/fs/lustre/lov.ko
insmod /lib/modules/3.10.0-
327.13.1.el7_lustre.x86_64/extra/kernel/fs/lustre/mdc.ko
insmod /lib/modules/3.10.0-
327.13.1.el7_lustre.x86_64/extra/kernel/fs/lustre/lmv.ko
insmod /lib/modules/3.10.0-
327.13.1.el7_lustre.x86_64/extra/kernel/fs/lustre/lustre.ko
```

Notice that the `lnet.ko` module is loaded as a dependency. The console and kernel ring buffer output will look something like this:

```
[266699.213610] LNet: HW CPU cores: 2, npartitions: 1
[266699.232630] alg: No test for adler32 (adler32-zlib)
[266699.234184] alg: No test for crc32 (crc32-table)
[266707.286906] Lustre: Lustre: Build Version: jenkins-
arch=x86_64,build_type=server,distro=el7,ib_stack=inkernel-40--
PRISTINE-3.10.0-327.13.1.el7_lustre.x86_64
[266707.338890] LNet: Added LNI 192.168.207.2@tcp [8/256/0/180]
[266707.339851] LNet: Accept secure, port 988
```

As can be seen in the above output, the LNet `networks` were automatically loaded.

The `lustre_rmmod` behavior is also different in this circumstance, compared to loading LNet on its own. If the administrator loads and configures LNet on its own, independently of the Lustre module, then it is necessary to unconfigure the LNet networks before removing the kernel modules:

```
[root@rh7z-pe ~]# modprobe lnet
[root@rh7z-pe ~]# lctl network up
LNET configured
[root@rh7z-pe ~]# lctl list_nids
192.168.207.2@tcp
[root@rh7z-pe ~]# lustre_rmmod
```

```
Modules still loaded:
lnet/klnds/socklnd/ksocklnd.o lnet/lnet/lnet.o
libcfs/libcfs/libcfs.o
[root@rh7z-pe ~]# lctl network down
LNET ready to unload
[root@rh7z-pe ~]# lustre_rmmod
[root@rh7z-pe ~]# lsmod |grep lnet
```

However, if the `lnet` module is loaded indirectly, as a dependency of the Lustre kernel module, then `lustre_rmmod` will gracefully unload all modules including `lnet`:

```
[root@rh7z-pe ~]# modprobe lustre
[root@rh7z-pe ~]# lctl list_nids
192.168.207.2@tcp
[root@rh7z-pe ~]# lustre_rmmod
[root@rh7z-pe ~]# lsmod | grep -E lnet\|lustre
```

This behavior is consistent, but not entirely intuitive. The reason for this behavior has to do with a special function of LNet: routing. LNet routing enables a node that is connected to more than one LNet fabric to route traffic between the networks. LNet routing is a complex topic and is not discussed in this guide. For more information on LNet routers, see:

http://www.intel.com/content/www/us/en/software/configuring-lnet-routers-file-systems-lustre-guide.html

Because routing is a function of the network, not of the Lustre file system itself, `lustre_rmmod` will effectively assume that if a host has only the `lnet` module loaded and running, then it is providing routing services. `lustre_rmmod` will therefore refuse to unload the modules unless the `lnet` service is explicitly unconfigured.

If, on the other hand, the `lustre` kernel module is also loaded, and there are no file systems mounted, then `lustre_rmmod` will assume that the host is either an idle server or client and will unload the entire stack, including the `lnet` modules.

If a Lustre OSD is mounted on a host, then the `lustre_rmmod` command will not unload the Lustre kernel modules and will report an error:

```
[root@rh7z-mds1 ~]# df -ht lustre
File system      Size  Used Avail Use% Mounted on
mgspool/mgt     960M  2.2M  956M   1% /lfs/mgt

[root@rh7z-mds1 ~]# lustre_rmmod
  0 UP osd-zfs MGS-osd MGS-osd_UUID 5
  1 UP mgs MGS MGS 7
```

```
  2 UP mgc MGC192.168.227.11@tcp1 c2108a9c-a62f-6626-48e4-
68f1caf1bce3 5
Modules still loaded:
lustre/mgs/mgs.o lustre/mgc/mgc.o lustre/quota/lquota.o
lustre/fid/fid.o lustre/fld/fld.o lnet/klnds/socklnd/ksocklnd.o
lustre/ptlrpc/ptlrpc.o lustre/obdclass/obdclass.o lnet/lnet/lnet.o
libcfs/libcfs/libcfs.o

[root@rh7z-mds1 ~]# df -ht lustre
File system      Size  Used Avail Use% Mounted on
mgspool/mgt     960M  2.2M  956M   1% /lfs/mgt
```

From this example, it can be seen that because the MGS is mounted, `lustre_rmmod` takes no action to remove the kernel modules. Instead, it shows that there are active services running on the host and exits. The MGT is still mounted and the MGS is running. The `lustre_rmmod` command is a very useful tool for ensuring the correct and safe unloading of Lustre kernel modules.

Optimizing o2iblnd Performance

In addition to defining the LNet interfaces, the kernel module files can be used to supply parameters to other kernel modules used by Lustre. This is commonly used to supply tuning optimizations to the LNet drivers, to maximize performance of the network interface. An example of this optimization can be seen in Lustre version 2.8.0 and the Intel® EE for Lustre* software version 3.0 and later, in the file `/etc/modprobe.d/ko2iblnd.conf`, which includes the following:

```
alias ko2iblnd-opa ko2iblnd
options ko2iblnd-opa peer_credits=128 peer_credits_hiw=64
credits=1024 concurrent_sends=256 ntx=2048 map_on_demand=32
fmr_pool_size=2048 fmr_flush_trigger=512 fmr_cache=1
```

This configuration is automatically applied to the LNet kernel module when an Intel Omni-Path interface is installed, but not when a different network interface is present.

The following set of options has been defined to optimize the performance of Intel® Omni-Path Architecture. A detailed description is beyond the scope of this exercise, but the following summary provides an overview:

- `peer_credits=128` - the number of concurrent sends to a single peer
- `peer_credits_hiw=64` - Hold in Wait – when to eagerly return credits
- `credits=1024` - the number of concurrent sends (to all peers)

- `concurrent_sends=256` - send work-queue sizing
- `ntx=2048` - the number of message descriptors that are pre-allocated when the ko2iblnd module is loaded in the kernel
- `map on demand=32` - the number of noncontiguous memory regions that will be mapped into a virtual contiguous region
- `fmr_pool_size=2048` - the size of the Fast Memory registration (FMR) pool (must be >= ntx/4)
- `fmr_flush_trigger=512` - the dirty FMR pool flush trigger
- `fmr_cache=1` - enable FMR caching

The default values used by Lustre if no parameters are given is:

- `peer_credits=8`
- `peer_credits_hiw=8`
- `concurrent_sends=8`
- `credits=64`

Optimizations are applied automatically on detection of an Intel® high performance network interface. Some of the parameters, such as FMR, are incompatible with other devices, such as Mellanox InfiniBand products using the MLX5 driver. It can be disabled by setting `map_on_demand=0` (the default). The configuration file can be modified or deleted to meet the specific requirements of a given installation.

In general, the default `ko2iblnd` settings work well with Mellanox InfiniBand HCAs and no tuning is normally required. Architecture differences between Intel® fabrics and Mellanox mean that setting universal defaults is very difficult. Intel® OPA and Intel® True Scale Fabric have an architecture that favors lightweight, high-frequency message-passing communications, compared to Mellanox, which has historically placed an emphasis on throughput-oriented workloads. Because Mellanox InfiniBand has historically been the dominant high-speed fabric, LNet driver development has naturally tended in the past to align with this technology, aided by interfaces that are intended to support storage-like workloads. What the above settings do is tune the LNet driver for communications on Intel® fabrics, if present.

Note: It is possible to use the `socklnd` driver on RDMA fabrics if there is an upper-level protocol that supports TCP/IP traffic, such as the IPoIB driver for InfiniBand fabrics. This use of `socklnd` on InfiniBand, RoCE, and Intel® OPA networks is *not* recommended because it will compromise the performance of LNet compared to the RDMA-based `o2iblnd`, and can have a negative impact on the stability of the resulting network connection. Instead, it is strongly

recommended that `o2iblnd` is used wherever possible; it provides the highest performance with the lowest overheads on these fabrics.

Dynamic LNet Configuration and lnetctl

In versions of Lustre up to and including Lustre version 2.6, if a modification to the LNet configuration is required, the modules need to be unloaded from the running kernel and then reloaded to pick up the changes. Because the Lustre kernel module depends on LNet for communication, the `lnet` module cannot be unloaded without also affecting the Lustre kernel module. Because Lustre is a network-based file service, this effectively means that the whole Lustre software stack for the affected host has to be stopped and then restarted.

Administrators of Lustre file systems will be most familiar with using the modprobe interface to configure LNet devices (described in an earlier section), and this remains the most common mechanism.

However, with the release of Lustre version 2.7.0, there is a new utility, called `lnetctl`, that offers a more flexible way to manage LNet configuration. With this new utility, LNet can also be updated while the kernel module is still running, a feature referred to as Dynamic LNet Configuration (DLC). Amongst other things, Dynamic LNet means tuning can be applied to an LNet interface on a host without incurring an outage.

The `lnetctl` utility can completely replace the older `modprobe` configuration method, but it is not mandatory; administrators can choose the tools that best match their needs. `lnetctl` is straightforward to use and helps to guide administrators to a valid, working configuration. The `lnetctl` man page has a comprehensive description of the configuration commands and options, as does the Lustre Operations Manual.

Dynamic LNet configuration is powerful and versatile, and provides administrators with easy tools to view and alter the running configuration of LNet on a host.

The LNet modules need to be loaded into the kernel prior to making any configuration changes using `lnetctl`. To load the lnet modules, use the `modprobe` command:

```
modprobe [-v] lnet
```

For example:

```
[root@rh7z-pe ~]# modprobe -v lnet
insmod /lib/modules/3.10.0-
327.13.1.el7_lustre.x86_64/extra/kernel/net/lustre/libcfs.ko
insmod /lib/modules/3.10.0-
327.13.1.el7_lustre.x86_64/extra/kernel/net/lustre/lnet.ko
```

After the modules are loaded, run the `lnet configure` sub-command to initialize the LNet service in the kernel:

```
lnetctl lnet configure [--all]
```

If the `--all` flag is used, `lnetctl` will try to load any LNet interfaces referenced as either `networks` or `ip2nets` options in the modprobe configuration files. Otherwise, `lnetctl lnet configure` will not attempt to initialize any networks. When using Dynamic LNet Configuration to manage the LNet interface for a host, using the `--all` flag is not recommended unless one is migrating from a legacy configuration with complex module options defined.

If there is no other configuration on the server, LNet will have a reference to the loopback interface and nothing else. For example:

```
[root@rh7z-pe ~]# lnetctl net show
net:
    - net: lo
      nid: 0@lo
      status: up
```

If the LNet kernel module is not loaded, then `lnetctl` will report an error when any of the commands are executed:

```
[root@rh7z-pe ~]# lnetctl lnet configure
opening /dev/lnet failed: No such device
hint: the kernel modules may not be loaded
configure:
    - lnet:
          errno: -19
          descr: "LNet configure error: No such device"
```

After loading the kernel module and running the `lnetctl lnet configure` command, `lnetctl` can now be used to create new LNet interfaces on the host. The syntax is explained in the `lnetctl` man page, but the basic format is as follows:

```
lnetctl net add --net <lnet name> --if <net interface> [<options> …]
```

Here is a simple example that creates a new NID configuration for LNet `tcp1` using the TCP/IP sockets LND on `eth1`:

```
lnetctl net add --net tcp1 --if eth1
```

This next example creates a new NID for an InfiniBand device using the OFED LND:

```
lnetctl net add --net o2ib0 --if ib0
```

To delete a NID from the LNet configuration, use the `lctl net del` command:

```
lnetctl net del --net <lnet name>
```

For example:

```
lnetctl net del --net tcp1
```

To review the currently configured NIDs, use:

```
lnetctl net show [--verbose]
```

For example:

```
[root@rh7z-pe ~]# lnetctl net show
net:
    - net: lo
      nid: 0@lo
      status: up
    - net: tcp
      nid: 192.168.207.2@tcp
      status: up
      interfaces:
          0: eth1
```

When invoked without any options, the `lnetctl` command also provides a command shell for interactive configuration:

```
[root@rh7z-pe ~]# lnetctl
lnetctl > help

Available commands are:
   lnet
   route
   net
   routing
   set
   import
   export
   stats
   peer_credits
   help
```

```
   exit
   quit
For more help type: help command-name
lnetctl > net
net {add | del | show | help}
lnetctl >
```

Configuration applied by `lnetctl` will be lost when the `lnet` kernel module is unloaded or the host is rebooted. To preserve the configuration, it can be exported to a file so that it can be re-imported when the host is restarted.

To save an LNet configuration, use:

```
lnetctl export [{file}]
```

If a file is not specified, the configuration is directed to `<stdout>`. For example:

```
[root@rh7z-pe ~]# lnetctl export
net:
    - net: lo
      nid: 0@lo
      status: up
      tunables:
          peer_timeout: 0
          peer_credits: 0
          peer_buffer_credits: 0
          credits: 0
    - net: tcp1
      nid: 192.168.207.2@tcp1
      status: up
      interfaces:
          0: eth1
      tunables:
          peer_timeout: 180
          peer_credits: 8
          peer_buffer_credits: 0
          credits: 256
```

The output for the `export` command is the same as for `lnetctl net show --verbose`.

All `lnetctl` output is formatted in YAML (Yet Another Markup Language), a relatively simple data structure syntax that is intended to be portable and has the benefit of being

straightforward to interpret. Libraries for parsing YAML exist for a wide variety of programming languages.

The `export` command is a great way to migrate hosts that have been using the kernel module options files via modprobe to use Dynamic LNet configuration. If a host has a running configuration that was provided using kernel module options, it can be exported using `lnetctl` into the YAML syntax.

There is, of course, a corresponding `import` command for `lnetctl`:

```
lnetctl import [--add|--del|--show] {file}
```

The file used as input must be formatted in YAML. The `import` command will also parse YAML input from `<stdin>`. The following example shows a simple configuration in YAML syntax:

```
net:
    - net: tcp1
      interfaces:
          0: eth1
```

This YAML configuration is equivalent to the following, expressed as a `modprobe` option:

```
options lnet networks="tcp1(eth1)"
```

When a configuration is exported by `lnetctl`, it will also include host-specific information such as the NID, interface status and tunables. However, the basic configuration need only describe the LNet (e.g. tcp1, o2ib0) and the interfaces, as above.

Note: To provide an effective and simple means to audit a host's lnet configuration, consider using the `lnetctl --add` command to create the `lnet` configuration for the first time, then use `lnetctl export` to capture a persistent record. In that way, one can create a tailored audit per host to verify that the recorded and running configurations match.

The default behavior of the import command is to add the interfaces described in the configuration file. The following two commands are therefore equivalent:

```
lnetctl import --add {file}
lnetctl import {file}
```

The following transcript shows an example of how the `lnetctl import --add` command works:

```
[root@rh7z-pe ~]# modprobe lnet # load the lnet kernel module
[root@rh7z-pe ~]# lnetctl lnet configure # start the lnet service
```

```
[root@rh7z-pe ~]# lnetctl net show
net:
    - net: lo
      nid: 0@lo
      status: up
[root@rh7z-pe ~]# cat lnet.cf
net:
    - net: tcp1
      nid: 192.168.207.2@tcp1
      status: up
      interfaces:
          0: eth1
      tunables:
          peer_timeout: 180
          peer_credits: 8
          peer_buffer_credits: 0
          credits: 256
[root@rh7z-pe ~]# lnetctl import --add lnet.cf
[root@rh7z-pe ~]# lnetctl net show
net:
    - net: lo
      nid: 0@lo
      status: up
    - net: tcp1
      nid: 192.168.207.2@tcp1
      status: up
      interfaces:
          0: eth1
```

If an LNet NID is already configured, then `lnetctl import [--add]` will return an error. For example, the following transcript shows an attempt to import a configuration for `tcp1(eth1)` on a host where the NID is already configured:

```
[root@rh7z-pe ~]# lnetctl net show
net:
    - net: lo
      nid: 0@lo
      status: up
    - net: tcp
      nid: 192.168.207.2@tcp1
      status: up
      interfaces:
          0: eth1
[root@rh7z-pe ~]# cat lnet.cf
```

```
net:
    - net: tcp
      nid: 192.168.207.2@tcp1
      status: up
      interfaces:
          0: eth1
      tunables:
          peer_timeout: 180
          peer_credits: 8
          peer_buffer_credits: 0
          credits: 256
[root@rh7z-pe ~]# lnetctl import lnet.cf
add:
    - net:
          errno: -17
          descr: "cannot add network: File exists"
```

The kernel will also report an error to the ring buffer and system console:

```
Mar  7 04:06:38 rh7z-pe kernel: LNetError: 2678:0:(api-
ni.c:1252:lnet_startup_lndni()) Net tcp1 is not unique
```

Note that `lnetctl export` will include the loopback interface in the output that it generates. Because the loopback NID is always created by the LNet module, regardless of the supplied configuration, it is not needed in the exported file output and LNet will actually generate an error when an attempt is made to import a configuration that contains the loopback NID. The error will not prevent the rest of the configuration file from being processed, but it does add unneeded noise to the host's log files and can be disconcerting to users who are not expecting this behavior. Unfortunately, there is not an easy way to exclude interfaces from the exported configuration and the loopback NID has to be removed by hand.

The `import --del` command will attempt to delete any NIDS configured on the host that match the specification described in the YAML configuration file. If the command is successful, then it will not return any output to the shell, but there will be an entry logged to the system console and to syslog. For example, if a file called `lnet.cf` contained an entry for NID `192.168.207.2@tcp1`, then the command:

```
[root@rh7z-pe ~]# lnetctl import --del lnet.cf
```

...will create an entry in the kernel ring buffer and syslog similar to the following:

```
Mar  7 04:46:00 rh7z-pe kernel: LNet: Removed LNI 192.168.207.2@tcp1
```

Any mismatched entries will generate an error – that is, if there is an entry in the YAML configuration that does not match a configured NID on the host, `lnetctl` will return an error for each such entry:

```
[root@rh7z-pe ~]# lnetctl import --del ln-mismatch.cf
del:
    - net:
          errno: -22
          descr: "cannot delete network: Invalid argument"
```

The `lnetctl import --show` command is used to identify which of the interfaces defined in the configuration file are actually active LNet NIDs configured on the host. If there are no matching NIDs, `lnetctl import --show` will not return any output.

```
[root@rh7z-pe ~]# lnetctl net show
net:
    - net: lo
      nid: 0@lo
      status: up
[root@rh7z-pe ~]# cat lnet.cf
net:
    - net: tcp1
      nid: 192.168.207.2@tcp1
      status: up
      interfaces:
          0: eth1
      tunables:
          peer_timeout: 180
          peer_credits: 8
          peer_buffer_credits: 0
          credits: 256
[root@rh7z-pe ~]# lnetctl import --show lnet.cf
[root@rh7z-pe ~]# lnetctl import --add lnet.cf
[root@rh7z-pe ~]# lnetctl import --show lnet.cf
net:
    - net: tcp1
      nid: 192.168.207.2@tcp1
      status: up
      interfaces:
          0: eth1
```

In the above example, the `lnet` kernel module on the host initially has only the loopback NID configured. When `lnetctl import --show` is run for the first time, it does not find a NID on the host that corresponds to the configuration, so it does not return any matches in its

output. After the configuration is added to LNet, the second invocation of `lnetctl import --show` lists a match. The `lnetctl import --show` command is most useful when debugging more complex configurations with the YAML syntax.

LNet automated startup and shutdown using sysvinit or systemd

Lustre provides a script for managing startup and shutdown of LNet on a host. It is written for use with sysvinit, and follows the Linux Standards Base format.

RHEL 7 has a new system and service management application used to control the startup and shutdown of services called systemd. This new application replaces the sysvinit service that is used in RHEL 6 and previous generations of Red Hat's operating system.

Currently, there is no direct support in Lustre for systemd, insofar as there is no specific systemd service unit for Lustre or LNet, but the lnet `sysvinit` script can be used by systemd without modification. In fact, RHEL 7 still includes the `chkconfig` and `service` tools in support of legacy init scripts.

Systemd is more powerful and flexible than sysvinit, and is able, for example, to support parallelism in the system boot and shutdown processes in a way that the venerable `init` process was unable to accomplish. Naturally, `systemd` is also far more complex than sysvinit, but it does maintain backwards compatibility with the older init scripts. This is good news for developers and documentation writers that would otherwise feel the need to editorialize on the merits or otherwise of this new, arbitrarily complex replacement for something that was already working.

The LNet `init` script is provided as a convenience to Lustre users and is not essential to operation of a Lustre environment. It is able to detect whether or not a host is using Dynamic LNet and will automatically load a configuration if it is found. Otherwise, the lnet module will be loaded with a static configuration based on module options recorded in the `/etc/modprobe.d` directory, or with the lnet default configuration if no module options are found.

To load lnet on system boot:

```
chkconfig lnet on
```

To start the lnet module manually:

```
service lnet start
```

To stop lnet:

```
service lnet stop
```

Running lnet as a separate service in this way is most useful when the host is configured as an LNet router. This is because routers do not run any other Lustre services: they only require the LNet kernel module, not the full Lustre software stack.

The `lnet` init script is also recommended when Dynamic LNet Configuration (DLC, i.e., the `lnetctl` command) is used to configure the LNet interfaces of a host. In the DLC, starting the LNet service needs to be separated from the startup of Lustre services (whether for server or client), because DLC explicitly imports the LNet configuration using the `lnetctl import` command as a separate step *after* the module is loaded; the configuration is not imported at the same time as the module load, which is the case with static LNet configurations.

If `lnetctl` is available on the target, and a YAML configuration is stored in `/etc/sysconfig/lnet.conf`, then the init script will load and configure the `lnet` kernel module. The simplest way to create the configuration file, is to configure LNet, then use the `lnetctl export` command to save the configuration:

```
lnetctl export /etc/sysconfig/lnet.conf
```

It is not always necessary to explicitly start and stop LNet on system boot or shutdown, and in fact one of the benefits of using the old static kernel module options configuration via modprobe is that the LNet configuration is automatically included. This is most obviously useful when the host is running Lustre services in addition to Lnet; loading the Lustre kernel module will cause its dependencies to be loaded automatically as well, including the LNet module. So with a single command, one can load the entire Lustre software stack.

However, reliance on implicit dependencies can also have negative consequences. For example, fault diagnosis and auditing can be more complex, as the relationship between modules with respect to startup and shutdown sequences may be poorly defined. And it can be beneficial to an administrator to have some assurance that the networking is correctly configured and working before committing to start services that depend on the network.

Reliance on implicit dependencies can also affect shutdown. If an assumption is made about the shutdown sequence of services, then it is possible for a service to be stopped out of sequence, causing a dependency to hang on shutdown rather than cleanly exit. On some releases of the Open Fabrics Enterprise Distribution (OFED) InfiniBand driver software, there is a shutdown script that is by default scheduled to execute while the LNet driver is still loaded. For these releases, because LNet depends on the OFED module, the OFED module will not be be unloaded, but because the module is not be unloaded, the init script will not exit, causing the host to hang on shutdown or reboot. To resolve this issue, ensure that the LNet init script is scheduled to execute prior to other network driver init scripts during shutdown.

Multi-rail LNet Topologies

Lustre does not have comprehensive support for single-fabric, multi-rail networks, although LNet can be configured to take advantage of bonded network interfaces when presented as a single device by the underlying transport. Be aware that devices using OFED drivers or the in-kernel InfiniBand drivers will only support active-passive, or failover, network bonding, which means that only one physical interface is active at any one point in time.

As a consequence, while it is possible to use a single network interface to join multiple LNets, lnetctl does not allow the inverse: one cannot use lnetctl to join multiple network interfaces to a single LNet. This behavior is consistent with best practices for LNet, because multi-rail configurations can lead to inconsistent routing across interfaces configured in this way.

A host can have multiple independent LNet interfaces configured and connected to separate networks. This enables servers to be directly connected to multiple fabrics simultaneously, or for a Lustre client to mount file systems that have been presented over different fabrics. It is only when a host tries to connect multiple interfaces to the same fabric that the limitations on multi-rail apply.

Multiple InfiniBand connections on a single fabric and configured into a bonded interface are currently only supported for the purposes of improving fault-tolerance, not for increasing throughput.

The `ko2iblnd` LND provides support for InfiniBand network device bonding in an active-passive configuration, for the purposes of high availability (HA). Because the bonded interface is active-passive, there is no improvement in throughput performance, so the feature is only suitable for use in situations where service availability is a mandated requirement (mission-critical platforms).

With this form of bonding, the server actively uses one interface in the bonded group at a time. If the active interface fails, traffic fails over to the remaining interface in the bond group.

This form of InfiniBand bonding support is distinct from the use of bonded network interfaces with `ksocklnd`, which runs over TCP/IP sockets. For Ethernet devices, `socklnd` is used, whether for bonded network connections or single interfaces.

Enabling InfiniBand (o2ib) Bonding

To enable failover support in LNet for bonded InfiniBand (or other network interfaces supported by OFED), add the following option into the kernel modules configuration:

```
options ko2iblnd dev_failover=1
```

The common convention is to create files in the directory `/etc/modprobe.d` containing options for loadable kernel modules.

With this option enabled, one can refer to the bonded network interface in the LNet configuration. For example:

```
options lnet networks=o2ib0(bond1)
```

The following example, based on a RHEL / CentOS operating platform, illustrates a bonded network configuration for a Lustre system with two InfiniBand interfaces.

```
/etc/modprobe.d/lustre.conf:
alias ibbond bonding
options lnet networks=o2ib0(ibbond)
options ko2iblnd dev_failover=1

/etc/sysconfig/network-scripts/ifcfg-ibbond:
DEVICE=ibbond
BOOTPROTO=none
IPADDR=10.0.0.11
NETMASK=255.255.0.0
ONBOOT=yes
TYPE=Bonding
USERCTL=no
MTU=2044
BONDING_OPTS="mode=1 miimon=100 primary=ib0"

/etc/sysconfig/network-scripts/ifcfg-ib0:
DEVICE=ib0
USERCTL=no
ONBOOT=yes
MASTER=ibbond
SLAVE=yes
BOOTPROTO=none
TYPE=InfiniBand

/etc/sysconfig/network-scripts/ifcfg-ib1:
DEVICE=ib1
USERCTL=no
ONBOOT=yes
MASTER=ibbond
SLAVE=yes
BOOTPROTO=none
TYPE=InfiniBand
```

The `ibbond` alias name is arbitrary, but is more descriptive than e.g. `bond0`, which is useful when there are multiple networks that the host is connected to. It is common to encounter

installations where there are both bonded Ethernet and bonded IB interfaces on the same host.

Restrictions for Multi-rail LNet Topologies

Because LNet does not natively support multi-rail topologies, i.e., multiple network interfaces connected to the same subnet, attempts to assign two interfaces to the same LNet will fail.

For example:

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

The above configuration will cause a syntax error when the kernel module is loaded and an attempt is made to start the network. The following transcript shows the behavior when this unsupported configuration is attempted:

```
[root@rh7z-pe ~]# modprobe -v lnet
insmod /lib/modules/3.10.0-
327.13.1.el7_lustre.x86_64/extra/kernel/net/lustre/libcfs.ko
insmod /lib/modules/3.10.0-
327.13.1.el7_lustre.x86_64/extra/kernel/net/lustre/lnet.ko
networks="tcp0(eth0),tcp0(eth1)"
[root@rh7z-pe ~]# lctl network up
LNET configure error 22: Invalid argument
```

The kernel ring buffer will have a record of the error reported by the LNet driver, for example:

```
[root@rh7z-pe ~]# dmesg | tail -1
[ 6620.324053] LNetError: 111-1: Duplicate network specified: tcp
```

The kernel will also log the error in the syslog:

```
[root@rh7z-pe ~]# tail -1 /var/log/messages
Feb 21 21:11:28 rh7z-pe kernel: LNetError: 111-1: Duplicate network
specified: tcp
```

Similarly, one cannot specify multiple interfaces within the parentheses associated with an LNet LND. In the following example, only the first interface, `eth0`, will be used to create an NID for the host; the second parameter, `eth1`, will be ignored:

```
# eth0 inet 192.168.207.2/24
# eth1 inet 192.168.207.111/24
[root@rh7z-pe ~]# modprobe -v lnet
insmod /lib/modules/3.10.0-
327.13.1.el7_lustre.x86_64/extra/kernel/net/lustre/libcfs.ko
```

```
insmod /lib/modules/3.10.0-
327.13.1.el7_lustre.x86_64/extra/kernel/net/lustre/lnet.ko
networks="tcp0(eth0,eth1)"
[root@rh7z-pe ~]# lctl network up
LNET configured
[root@rh7z-pe ~]# lctl list_nids
192.168.207.2@tcp
```

LNet Configuration Edge Case Behaviors and Side-Effects

If there is no configuration defined, either through a `modprobe` options file or a YAML description for `lnetctl`, LNet will create a NID on the first network interface detected by the OS (usually `eth0`) when the `lnet` kernel module is loaded and the LNet service is brought online with `lctl network up`.

The `ip2nets` option for the LNet kernel module is a list of network definition and IP-match pairs. These pairs are processed in sequence. If there is a match for a local IP address, then that network definition is used for the node, and further pairs for that network are ignored. Multiple networks can be matched.

For example:

```
ip2nets="tcp(eth2) 134.32.1.[4-10/2]; tcp(eth1) *.*.*.*"
```

This set of rules is used to create network tcp0 (the 0 is implied, since the LNet network number is omitted). If a local IP address matches 134.32.1.[4-10/2], meaning it is one of `134.32.1.4`, `134.32.1.6`, `134.32.1.8`, or `134.32.1.10`, then `tcp0` is created using interface `eth2`. Otherwise the second pair is used, and because `"*.*.*.*"` matches every address, it always creates `tcp0` on `eth1`.

Note that ip2nets will use the IP address definition to match the host, not the interface. The ip2nets definition will not verify or otherwise qualify that the IP address matched is associated with the physical network interface in the specification. This means that a pattern can match the IP address of an interface that will not actually be used for LNet communications. From the above example, if a host has an interface eth3 with IP address 134.32.1.4, then that would be considered a match good enough to trigger the creation of the NID on `tcp0(eth2)`.

Also, if the device is not specified in an ip2nets definition, LNet will pick up the first available device rather than the device that matches the IP address pattern. For example, if the IP address pattern matches the IP address on eth1, but no device is mentioned in the ip2nets definition, then eth0 will get an LNet configuration. As an illustration, consider a host with a 10.70/16 IP address on eth0, and a 192.168/24 address on eth1. The following ip2nets definition will create a NID on eth0, even though the pattern matches the eth1 device:

```
options lnet ip2nets="tcp0 192.168.*.*"
```

If, instead, the device is included in the spec, then the configuration will be applied to eth1:

```
options lnet ip2nets="tcp0(eth1) 192.168.*.*"
```

The definition is interpreted as follows: configure the first socklnd NID that is found on the host where there is an IP Address matching `192.168.*.*`. In this respect, it's consistent with the behavior of the much simpler networks syntax in the following example:

```
options lnet networks="tcp0"
```

This example creates a NID on the first network device detected by the operating system, because no device was specified. In common with the ip2nets parameter, the lack of definition of a specific network interface means that LNet will configure the first interface that was detected by the host operating system.

If an interface is explicitly specified as well as a pattern, the interface matched using the IP pattern will be compared against the explicitly defined interface. For example, if the `ip2nets` definition is "`tcp(eth0) 192.168.*.3`" and there exists in the system a device `eth0` with IP address `192.0.19.3` and a device `eth1` with IP address `192.168.3.3`, then configuration will fail, because the pattern contradicts the interface specified. A clear warning will be displayed if inconsistent configuration is encountered.

If the LNet number for a NID is 0 (zero), for example, tcp0, or o2ib0, the number will sometimes be omitted from command output, and can usually be omitted from configuration files as well (although it is not recommended -- for reasons of clarity alone, it is recommended to supply as much information as is reasonable when creating configuration information).

Lustre Storage Devices

All persistent information for a Lustre file system is contained on block storage file systems distributed across a set of storage servers. Lustre's architecture is built on a distributed object storage model where back-end block storage is abstracted by an API called the Object Storage Device, or OSD. The OSD enables Lustre to use different back-end file systems for persistence, of which LDISKFS (based on EXT4) and ZFS are the two currently supported. The term OSD is also used as a generic term for an instance of a Lustre storage target, such as an MGT, MDT or OST. For example, if an action can be applied to any storage target, the term OSD will normally be used, rather than writing out all three types of storage.

Lustre objects can either be data objects holding a byte stream for file data, or index objects, which are typically used for metadata such as directory information.

A single OSD instance corresponds to precisely one back-end storage volume. Physical devices are assembled into logical units or volumes and these are used to create OSD instances. Lustre has three types of OSD instance, corresponding to the types of Lustre services. These are:

- Management Target (MGT): used by the Management Service (MGS) to maintain file system configuration data used by all hosts in the Lustre environment.
- Metadata Target (MDT): used by the Metadata Service (MDS) to record the file system name space (file and directory information for an instance of Lustre)
- Object Storage Target (OST): used by the Object Storage Service (OSS) to record data objects representing the contents of files.

The term OSD can also used as a generic term for a physical Lustre object store, in place of MGT, MDT or OST. In this case, what is meant is any storage device or LUN that has been formatted with Lustre on-disk data structures.

Formatting Lustre Storage

Lustre storage has to be formatted for each supported service: MGT, MDT or OST. The mkfs.lustre command is supplied with the Lustre server software for this purpose. The general command syntax is as follows:

```
mkfs.lustre { --mgs | --mdt | --ost } \
  [ --reformat ] \
  [ --fsname <name> ] \
  [ --index <n> ] \
  [ --mgsnode <MGS NID> [--mgsnode <MGS NID> …]] \
  [ --servicenode <NID> [--servicenode <NID> …]] \
  [ --failnode <NID> [--failnode <NID> …]] \
```

```
[ --backfstype=zfs ] \
[ --mkfsoptions <options> ] \
{ <pool name>/<dataset> [<zpool specification>]] | <device> }
```

Formatting storage targets is covered in detail later on; for now, just be aware that all Lustre OSDs are created using the same `mkfs.lustre` command-line application.

The purpose of the storage target is determined at format time through selection of one of `--mgs`, `--mdt` or `--ost`. The software also defines the extent to which a storage target is to be made highly-available, through the `--failnode` or `--servicenode` flags. For MDTs and OSTs, the administrator must supply a name (`--fsname`) for the Lustre file system to which they are allocated, and an index number (`--index`), which is a unique non-negative integer within the file system storage population. Additional formatting options can be supplied to the underlying backing store file system with the `--mkfsoptions` parameter; EXT4 and ZFS file system datasets options can be supplied, but note that zpool options cannot be included directly using `mkfs.lustre`.

When working with ZFS-based storage, one can use the `mkfs.lustre` command to assemble the ZFS pools, and also create the file system datasets that will contain the Lustre on-disk data. In this case, the ZFS pool specification is supplied along with the pool name and dataset name. However, as will be explained later in this chapter, this approach is not always suitable when working with production configurations because it does not allow an administrator to set or override properties of the ZFS pool.

Defining Service Failover (`--failnode` vs `--servicenode`)

All Lustre file system storage services are associated with block storage targets that contain a set of data for a given file system. The content varies by service type, but the sum of all of the data on all of the storage targets represents each file system as a whole. Loss of access to a service, for example because the host running the service has crashed, effectively means loss of access to the associated storage targets and the data they contain.

Because data is not replicated in Lustre (due to the large performance and latency overheads that replication entails), loss of a server can mean loss of access to data unless some other provision is made to ensure continuity. To protect against server loss, Lustre makes use of a standard high availability paradigm called failover, whereby a service can be run on one of a set of hosts, and if it fails, one of the other hosts in the set can restart the services that were running on the host that failed. For this to work with Lustre, the storage targets must be accessible by each host in the failover group. This is accomplished by connecting the servers to one or more arrays of stand-alone, shared storage (i.e., storage that is external to the server and contained in its own chassis).

Lustre services are usually grouped into HA pairs – two servers connected to a common pool of shareable block storage.

Failover is an attribute of each Lustre storage target, and is written into each storage target's configuration. Each target contains a list of the host NIDs that are able to mount the storage and present it to the network. This list of NIDs is registered with the MGS on mount, so that the clients know which NIDs to connect to. If one of the NIDs does not respond, the client service will try the next NID in the list for that service. When it runs out of unique NIDs, the client will retry the list in order from the top until a connection is made. By using failover with shared storage, Lustre is resilient to failures and clients are able to tolerate outages in server infrastructure.

There are two ways to define the failover configuration for a Lustre storage target:

1. `failnode`: this is the original method for defining failover groups, and it creates a configuration where there is a primary or preferred host for a given target, and one or more failover hosts.
2. `servicenode`: this is a newer option for defining the set of hosts where a given target can be mounted. With `servicenode`, there is no defined primary node for a service. Notionally, all hosts are equally able to run a given service, with no defined preferred primary. This is the recommended method for defining failover.

Each method is valid and supported by Lustre, but the methods are mutually incompatible. A storage target can contain either a failnode configuration or a servicenode configuration, but not both.

Using the failnode configuration syntax, the administrator lists the set of failover nodes that a storage target can be accessed from, but does not explicitly define the primary node. The primary node is not determined until the first time that a formatted storage target is mounted, at which point the configuration is updated with the NID of the server where the mount command is executed.

This means that the primary node must be online and available for service when the storage target is mounted the first time. It also means that there is potential for mistakes to creep into process execution, because the failnode configuration is dependent on a very specific start-up sequence for the first time mount of a Lustre device.

Also note that the `tunefs.lustre` command will only list the hosts listed as "failnodes" in the command line, and this does not get updated with information about the primary node, even after the OSD is mounted. For example:

```
[root@rh7z-mds1 ~]# mkfs.lustre --reformat --mgs   --failnode
192.168.227.12@tcp1   --backfstype=zfs   mgspool/mgt

   Permanent disk data:
Target:     MGS
```

```
Index:      unassigned
Lustre FS:
Mount type: zfs
Flags:      0x64
              (MGS first_time update )
Persistent mount opts:
Parameters: failover.node=192.168.227.12@tcp1

mkfs_cmd = zfs create -o canmount=off -o xattr=sa mgspool/mgt
Writing mgspool/mgt properties
  lustre:version=1
  lustre:flags=100
  lustre:index=65535
  lustre:svname=MGS
  lustre:failover.node=192.168.227.12@tcp1
[root@rh7z-mds1 ~]# tuner^C
[root@rh7z-mds1 ~]# ^C
[root@rh7z-mds1 ~]# tunefs.lustre --dryrun mgspool/mgt
checking for existing Lustre data: found

   Read previous values:
Target:     MGS
Index:      unassigned
Lustre FS:
Mount type: zfs
Flags:      0x44
              (MGS update )
Persistent mount opts:
Parameters: failover.node=192.168.227.12@tcp1


   Permanent disk data:
Target:     MGS
Index:      unassigned
Lustre FS:
Mount type: zfs
Flags:      0x44
              (MGS update )
Persistent mount opts:
Parameters: failover.node=192.168.227.12@tcp1

exiting before disk write.
```

The `servicenode` syntax defines a list of peers equally capable of mounting the storage target and there is no implied primary host. Any one of the defined peers can mount the storage and start the services for that storage. This method is much easier to implement and maintain, because all server NIDs are written directly into the configuration – there is no ambiguity about which hosts are associated with the storage target.

It is recommended that the `servicenode` method be used for creating HA failover configurations for Lustre storage targets, given the increased flexibility in adding new services, explicit definition of the hosts that are able to run the service, and the ability to better exploit the resource management features of HA software frameworks like Pacemaker.

Lustre Device and Mount Point Naming Conventions

There are three device types for Lustre storage: MGT, MDT and OST. These correspond to the MGS, MDS and OSS Lustre services, respectively. The following guidance has been developed as a recommended naming convention for Lustre's persistent storage components:

Service Name	ZFS Pool Name*	ZFS Dataset Name*	Mount Point
MGS	`mgspool`	`mgt`	`/lustre/mgt`
MDS	`<fsname>mdt<n>pool`	`mdt<n>`	`/lustre/<fsname>/mdt<n>`
OSS	`<fsname>ost<n>pool`	`ost<n>`	`/lustre/<fsname>/ost<n>`
Client	`n/a`	`n/a`	`/lustre/<fsname>`

* ZFS Pool Name and Dataset name only apply to ZFS-based storage targets.

Note that the Lustre file system name is limited to eight characters. Yes, like DOS.

The naming of pools, datasets and mount points presented here is provided as a recommendation only. Administrators can make their own choices about naming. The convention chosen here has been designed to provide a standard for describing the components that is unambiguous and easy to interpret.

The MGT is the persistent data store for the MGS, which is a global resource, and the only Lustre service that is independent of any specific Lustre file system. As such, the ZFS dataset name and file system mount point do not make reference to a Lustre file system instance.

All other Lustre storage devices should make reference to the file system name.

ZFS OSDs

When working with ZFS OSDs, one can bundle the entire process into a single command using `mkfs.lustre`, or split the work into two smaller, more finely-controlled steps where

creation of the zpool is separated from formatting the OSD. Both methods are discussed in this section, however we recommend creating the ZFS storage pools separately from formatting the Lustre OSD. By separating these tasks, it is easier to apply tuning options to ZFS, and it becomes clearer which options affect ZFS and which affect Lustre.

Furthermore, for high-availability configurations where the ZFS volumes are kept on shared storage, the zpools must be created independently of the `mkfs.lustre` command in order to be able to correctly prepare the zpools for use in a high-availability, failover environment.

ZFS Storage Pool Basics

ZFS separates storage volume definition from the file system specification, providing two separate tools to manage each. The zpool command is used to define the volumes and manages the physical storage assets, while the zfs command provides management of the ZFS file system datasets themselves.

A ZFS pool is comprised of one or more entities called Virtual Devices or vdevs. There are two basic categories of vdev: physical and logical. A physical vdev can be a complete physical storage device such as a disk drive, a partition on a disk drive, or a file. For Lustre file systems, it is strongly recommended that whole disks are used, with no pre-defined partition table.

Logical vdevs are assemblies of physical vdevs, arranged into groups, usually for the purpose of providing additional storage redundancy. Logical vdevs include mirrors and RAIDZ data protection layouts. There can be many vdevs assigned per ZFS pool, and data is written in stripes across the vdevs in the pool. The following examples illustrate how ZFS pools are created.

Simple stripe across two physical vdevs:

```
zpool create tank sda sdb
```

The result of this command is the creation of a pool named `tank` containing two physical vdevs, `sda` and `sdb`, in a stripe (equivalent to RAID 0).

Two-disk mirror:

```
zpool create tank mirror sda sdb
```

Striped mirrors (equivalent to RAID 1+0):

```
zpool create tank mirror sda sdb mirror sdc sdd mirror sde sdf
```

The above example has a single pool consisting of three mirrored vdevs. Data is striped across the three mirrors.

Pool with single RAIDZ2 vdev (equivalent to RAID 6):

```
zpool create tank raidz2 sda sdb sdc sdd sde sdf
```

Pool with two RAIDZ2 vdevs (equivalent to RAID 6+0):

```
zpool create tank raidz2 sda sdb sdc sdd sde sdf \
  raidz2 sdg sdh sdi sdj sdk sdl
```

Formatting a ZFS OSD using only the `mkfs.lustre` command

The basic syntax for creating a ZFS-based OSD using only the `mkfs.lustre` command is as follows:

```
mkfs.lustre --mgs | --mdt | --ost \
  [--reformat] \
  [ --fsname <name> ] \
  [ --index <n> ] \
  [ --mgsnode <MGS NID> [--mgsnode <MGS NID> …]] \
  [ --servicenode <NID> [--servicenode <NID> …]] \
  [ --failnode <NID> [--failnode <NID> …]] \
  --backfstype=zfs \
  [ --mkfsoptions <options> ] \
  <pool name>/<dataset> \
  <zpool specification>
```

The `servicenode` and `failnode` command-line options are used to identify the NIDs of the hosts that are able to run the Lustre service in a high-availability configuration. The options `servicenode` and `failnode` are mutually incompatible: choose one or the other when defining the HA failover hosts that are expected to provide the Lustre service.

The `servicenode` syntax defines all of the NIDs of all of the hosts that will be able to run the Lustre service. All of the hosts must be referenced, including the host that is expected to be the preferred primary for running the service (this is usually the host where the format command is running).

This example uses the `--servicenode` syntax to create an MGT that can be run on two servers as an HA failover resource:

```
[root@rh7z-mds1 ~]# mkfs.lustre --mgs \
  --servicenode 192.168.227.11@tcp1 \
  --servicenode 192.168.227.12@tcp1 \
  --backfstype=zfs \
  mgspool/mgt mirror sda sdc
```

The command line formats a new MGT that will be used by the MGS for storage. The command further defines a mirrored zpool called `mgspool` consisting of two devices, and creates a ZFS dataset called `mgt`. Two server NIDs are supplied as service nodes for the MGS, `192.168.227.11@tcp1` and `192.168.227.12@tcp1`.

The `failnode` syntax is similar, but is used to define only a failover target for the storage service. The `failnode` syntax is an older method for creating services and implicit in the definition of the storage service is the notion of a primary server and one or more secondary, or failover servers. Only the failover servers are included in the command-line definition. The primary is only written to the storage service configuration the first time that the formatted device is mounted. Whichever host mounts the storage first will have its NID written in as the effective primary server.

Example format command with the `failnode` syntax:

```
[root@rh7z-mds1 ~]# mkfs.lustre --mgs \
  --failnode 192.168.227.12@tcp1 \
  --backfstype=zfs \
  mgspool/mgt mirror sda sdc
```

Here, the failover host is identified as `192.168.227.12@tcp1`, one MDS server in an HA pair (and which, incidentally, has the hostname `rh7z-mds2`). The `mkfs.lustre` command was executed on `rh7z-mds1` (NID: `192.168.227.11@tcp1`), and the `mount` command must also be run from this host when the MGS service starts for the very first time. Otherwise, the primary NID will not be written to the storage configuration, and the failover mechanism will not work as expected. Note that if after formatting the storage, MGT is mounted on `192.168.227.12@tcp1`, then both the primary NID and the failover NID would be the same. The intended primary host, `192.168.227.11@tcp1`, would be excluded from being able to run the MGS service.

Wherever possible, use the `servicenode` syntax to define the high availability configuration for Lustre services.

The `mkfs.lustre` command can pass additional command flags to the underlying file system creation software using the `--mkfsoptions` flag. The command was originally used to pass through options for EXT-based storage, but can also be used in a limited way for ZFS. The `--mkfsoptions` parameter allows a user to pass through commands that are added to the `zfs` command line utility, but cannot be used to modify the `zpool` command line invocation. There are times when the `zpool` command defaults are not sufficient to support a production Lustre file system, especially when the storage system is on shared drives and there is a failover configuration in place.

For this reason, it is recommended that ZFS pools always be created explicitly and separately from the `mkfs.lustre` command.

Formatting a ZFS OSD using **zpool** and **mkfs.lustre**

To create a ZFS-based OSD suitable for use as a high-availability failover storage device, first create a ZFS pool to contain the file system dataset, then use `mkfs.lustre` to actually create the file system inside the zpool:

```
zpool create [-f] -O canmount=off \
  [ -o ashift=<n> ] \
  -o cachefile=/etc/zfs/<zpool name>.spec | -o cachefile=none \
  <zpool name> <zpool specification>

mkfs.lustre --mgs | --mdt | --ost \
  [--reformat] \
  [ --fsname <name> ] \
  [ --index <n> ] \
  [ --mgsnode <MGS NID> [--mgsnode <MGS NID> …]] \
  [ --servicenode <NID> [--servicenode <NID> …]] \
  [ --failnode <NID> [--failnode <NID> …]] \
  --backfstype=zfs \
  [ --mkfsoptions <options> ] \
  <pool name>/<dataset name>
```

For example:

```
# Create the zpool
zpool create -O canmount=off \
  -o cachefile=none \
  mgspool mirror sda sdc

# Format the Lustre MGT
mkfs.lustre --mgs \
  --servicenode 192.168.227.11@tcp1 \
  --servicenode 192.168.227.12@tcp1 \
  --backfstype=zfs \
  mgspool/mgt
```

After formatting, use `tunefs.lustre` to review the newly created OSD. The command line format is:

```
tunefs.lustre --dryrun <pool name>/<dataset name>
```

For example:

```
[root@rh7z-mds1 ~]# tunefs.lustre --dryrun mgspool/mgt
checking for existing Lustre data: found
```

```
   Read previous values:
Target:     MGS
Index:      unassigned
Lustre FS:
Mount type: zfs
Flags:      0x1044
              (MGS update no_primnode )
Persistent mount opts:
Parameters: failover.node=192.168.227.11@tcp1:192.168.227.12@tcp1


   Permanent disk data:
Target:     MGS
Index:      unassigned
Lustre FS:
Mount type: zfs
Flags:      0x1044
              (MGS update no_primnode )
Persistent mount opts:
Parameters: failover.node=192.168.227.11@tcp1:192.168.227.12@tcp1
```

Use the `zfs get` command to retrieve comprehensive metadata information about the file system dataset and to confirm that the Lustre properties have been set correctly:

```
zfs get all | awk '$2 ~ /lustre/'
```

For example:

```
[root@rh7z-mds1 ~]# zfs get all | awk '$2 ~/lustre/'
mgspool/mgt  lustre:version        1                                          local
mgspool/mgt  lustre:index          65535                                      local
mgspool/mgt  lustre:failover.node  192.168.227.11@tcp1:192.168.227.12@tcp1  local
mgspool/mgt  lustre:svname         MGS                                        local
mgspool/mgt  lustre:flags          4196                                       local
```

Only the zpool is created directly by the administrator. The `mkfs.lustre` command is still used to control creation of the file system dataset from the pool. Additional properties of the data set can be applied by `mkfs.lustre` using the `--mkfs.options` flag. The `mkfs.lustre` command will fail with an error if an attempt is made to format an existing dataset:

```
[root@rh7z-mds1 ~]# mkfs.lustre --mgs \
>  --servicenode 192.168.227.11@tcp1 \
>  --servicenode 192.168.227.12@tcp1 \
>  --backfstype=zfs   mgspool/mgt
```

```
   Permanent disk data:
Target:     MGS
Index:      unassigned
Lustre FS:
Mount type: zfs
Flags:      0x1064
              (MGS first_time update no_primnode )
Persistent mount opts:
Parameters:  failover.node=192.168.227.11@tcp1:192.168.227.12@tcp1

checking for existing Lustre data: not found
mkfs_cmd = zfs create -o canmount=off -o xattr=sa mgspool/mgt
   cannot create 'mgspool/mgt': dataset already exists

mkfs.lustre FATAL: Unable to create file system mgspool/mgt (256)

mkfs.lustre FATAL: mkfs failed 256
```

One can force the dataset to be formatted as a Lustre OSD by adding the `--reformat` flag to `mkfs.lustre`:

```
[root@rh7z-mds1 ~]# mkfs.lustre --reformat --mgs \
>  --servicenode 192.168.227.11@tcp1 \
>  --servicenode 192.168.227.12@tcp1 \
>  --backfstype=zfs   mgspool/mgt

   Permanent disk data:
Target:     MGS
Index:      unassigned
Lustre FS:
Mount type: zfs
Flags:      0x1064
              (MGS first_time update no_primnode )
Persistent mount opts:
Parameters:  failover.node=192.168.227.11@tcp1:192.168.227.12@tcp1

mkfs_cmd = zfs create -o canmount=off -o xattr=sa mgspool/mgt
Writing mgspool/mgt properties
  lustre:version=1
  lustre:flags=4196
  lustre:index=65535
  lustre:svname=MGS
  lustre:failover.node=192.168.227.11@tcp1:192.168.227.12@tcp1
```

Alternatively, destroy the dataset and have `mkfs.lustre` recreate it:

```
zfs destroy <pool name>/<dataset name>

mkfs.lustre --mgs \
  [ --servicenode <NID> [--servicenode <NID> …]] \
  --backfstype=zfs \
  [ --mkfsoptions <options> ] \
  <pool name>/<dataset>
```

If the dataset is reformatted, then previously applied properties will obviously be lost. Remember to include any ZFS-specific properties by making use of the `--mkfsoptions` flag.

Working with ZFS Imports

The assembly and incorporation of a ZFS storage pool into the operating system run-time environment is managed by a command called `zpool import`. Existing pools are incorporated into a host's run-time environment using the `zpool import` command, and can be released using the `zpool export` command. The basic syntax to import a pool is:

```
zpool import [-f] [ -o <properties>] <pool name>
```

This technique enables storage pools to migrate between hosts in a consistent and reliable manner. Any host that is connected to a pool of storage in a shared enclosure can import the ZFS pool, making it straightforward to facilitate high availability failover.

A ZFS storage pool can only be imported to a single host at a time. If a pool has been imported onto a host, it must be exported before it can be safely imported to a different host. If a zpool in a shared storage enclosure is simultaneously imported to more than one host, the pool data will be corrupted. To reduce the risk of this happening, all servers must be configured with a unique hostid that is used to label each zpool with the current active host that has imported the pool. Please see "Protecting File System Volumes from Concurrent Access (Multi-mount Protection)" for how to enable and verify that this protection is properly configured.

If the pool is not exported, it will appear to still be active on its original host, even if that host is offline (regardless of whether it was powered off cleanly or crashed). In this case, the import will fail, provided that the hostids for the servers have been correctly configured, and the import will need to be forced.

If an import fails, it could mean that the pool is already imported on a different host. Before trying to force the importation of a pool onto a host, check that the storage has not already been imported to another system. If the zpool has been configured correctly, and all hosts

have valid "hostids", then the import command will indicate the host that last had the pool imported. The remainder of this chapter will outline ways to verify the status of a ZFS pool.

The `zpool import` command can be used to list pools that are not currently imported on the host, without actually performing any import:

```
zpool import [-d <dev directory>] [-D]
```

When invoked using this syntax, `zpool import` lists the pools that are potentially available for import, and will ignore zpools that are already imported to the host. The `-d` (lower case) flag is used to specify a directory containing block devices, for example `/dev/disk/by-id`. This flag is not often required. The `-D` (upper case) flag is used to list zpools that have been destroyed.

In the following example, the import fails because the zpool cannot be found:

```
[root@rh7z-mds2 system]# zpool import demo-mdtpool
cannot import 'demo-mdtpool': no such pool available
```

This could be because the zpool does not exist, or because the pool has the wrong name, but it could also mean that the pool has been destroyed. To check, run `zpool import` without any other options to get a list of the zpools that the operating system can locate, that are not already imported to the host:

```
[root@rh7z-mds2 system]# zpool import
   pool: mgspool
     id: 2186474330384511828
  state: ONLINE
 status: The pool was last accessed by another system.
 action: The pool can be imported using its name or numeric
identifier and
   the '-f' flag.
   see: http://zfsonlinux.org/msg/ZFS-8000-EY
 config:

   mgspool     ONLINE
     mirror-0  ONLINE
       sda     ONLINE
       sdc     ONLINE
```

The output only lists a single zpool, called `mgspool`. There is no obvious sign of any other configured zpool available to the operating system. If the list of ZFS pools does not match expectations, perhaps one or more of the pools has been deleted. Check using `zpool import -D`:

```
[root@rh7z-mds2 system]# zpool import -D
   pool: demo-mdtpool
     id: 6641674394771267657
  state: ONLINE (DESTROYED)
 action: The pool can be imported using its name or numeric
identifier.
 config:

   demo-mdtpool                                    ONLINE
     mirror-0                                    ONLINE
       scsi-0QEMU_QEMU_HARDDISK_EEMDT0001  ONLINE
       scsi-0QEMU_QEMU_HARDDISK_EEMDT0000  ONLINE
```

From this, it can be seen that at some point in the past, another zpool existed on the system, but that it was destroyed. It is possible to recover a destroyed pool as follows:

```
zpool import -D <pool name>
```

Provided that the storage from which the pool was originally assembled has not been modified or the data over-written, the pool will be re-assembled and can be used as normal.

The `zdb` command can be helpful in determining where a zpool has been imported. If an exported pool cannot be imported cleanly into a host, use `zdb` to check the MOS configuration to see if it is perhaps "registered" with another host:

```
zdb -e <zpool name> | awk '/^MOS/,/^$/{print}'
```

The `hostid` and `hostname` fields will indicate the last host known to have imported the pool or dataset. For example:

```
[root@rh7z-mds1 ~]# zdb -e mgspool | awk '/^MOS/,/^$/{print}'
MOS Configuration:
        version: 5000
        name: 'mgspool'
        state: 0
        txg: 34351
        pool_guid: 11089712772589408485
        errata: 0
        hostid: 1386610045
        hostname: 'rh7z-mds2'
        vdev_children: 1
        vdev_tree:
…
<output truncated for brevity>
```

In this example, it would be prudent to check the host `rh7z-mds2` to see if the pool has been imported there before taking any further action.

Lustre and ZFS File System Datasets

When working with ZFS-based storage, each Lustre storage target is held on a file system dataset inside a ZFS pool. The dataset will be created by Lustre when the storage is formatted with the `mkfs.lustre` command and `--backfstype=zfs` has been selected. While it is possible to create multiple file system datasets within a single storage pool and use those for Lustre, this is not recommended: each dataset will compete for the pool's resources, affecting performance and making it more difficult to balance IO across the storage cluster. Also bear in mind that the unit of failover is the ZFS pool, not the dataset. One cannot migrate a dataset in a pool without migrating the entire pool. For example, if the MGT and MDT0 are created within the same ZFS pool, then the MGS and MDS services will always have to run on the same host, because the pool can only be imported to one host at a time. The MGS therefore loses its independence from the MDS for MDT0.

Don't use the `zfs` command directly to create datasets that will be used as Lustre targets. ZFS datasets created independently of the `mkfs.lustre` command will have be unmounted or destroyed and then reformatted. The properties of Lustre file system datasets can be altered after formatting or can be supplied as options to the `mkfs.lustre` command by using the `--mkfsoptions` flag. Refer to the `mkfs.lustre` manual page for details.

If a ZFS dataset already exists and is not unmounted, the `mkfs.lustre` command will not report an error when an attempt is made to format, but it will not be able to format the volume. The only immediate indication of a failure is that the `mkfs.lustre` output will be truncated:

```
[root@rh7z-mds1 ~]# zpool create -O canmount=off -o cachefile=none
mgspool mirror sda sdc

# Create a file system dataset using the default properties.
# The dataset will be automatically mounted once created.
[root@rh7z-mds1 ~]# zfs create mgspool/mgt

# Try to format the dataset for Lustre.
# The command will fail but will not report an error.
[root@rh7z-mds1 ~]# mkfs.lustre --mgs \
>   --servicenode 192.168.227.11@tcp1 \
>   --servicenode 192.168.227.12@tcp1 \
>   --backfstype=zfs \
>   mgspool/mgt
```

```
   Permanent disk data:
Target:     MGS
Index:      unassigned
Lustre FS:
Mount type: zfs
Flags:      0x1064
              (MGS first_time update no_primnode )
Persistent mount opts:
Parameters:  failover.node=192.168.227.11@tcp1:192.168.227.12@tcp1
[root@rh7z-mds1 ~]#
```

Administrators must use the `mount.lustre` command whenever starting Lustre services, and the standard `umount` command when stopping services.

For the purposes of comparison, let's examine what the `mkfs.lustre` command itself is doing when it creates/formats a ZFS OSD. The following example command output shows an MGT being created from a ZFS mirrored zpool consisting of two disks:

```
[root@rh7z-mds1 ~]# mkfs.lustre --mgs \
>  --servicenode 192.168.227.11@tcp1 \
>  --servicenode 192.168.227.12@tcp1 \
>  --backfstype=zfs   mgspool/mgt

   Permanent disk data:
Target:     MGS
Index:      unassigned
Lustre FS:
Mount type: zfs
Flags:      0x1064
              (MGS first_time update no_primnode )
Persistent mount opts:
Parameters:  failover.node=192.168.227.11@tcp1:192.168.227.12@tcp1

checking for existing Lustre data: not found
mkfs_cmd = zfs create -o canmount=off -o xattr=sa mgspool/mgt
Writing mgspool/mgt properties
  lustre:version=1
  lustre:flags=4196
  lustre:index=65535
  lustre:svname=MGS
  lustre:failover.node=192.168.227.11@tcp1:192.168.227.12@tcp1
[root@rh7z-mds1 ~]#
```

Note that the `mkfs.lustre` command hands creation of the `mgspool` zpool and `mgt` dataset to the relevant ZFS commands, acting as a convenient wrapper that encapsulates creation of the ZFS dataset and the Lustre formatted storage devices. The complete `zpool` and `zfs` command-line invocations are displayed in the `mkfs.lustre` output, providing transparency in the way that the underlying storage is configured.

Unfortunately, the `zpool` command line options used by `mkfs.lustre` cannot be modified, other than to define the zpool specification. The `mkfs.lustre` command therefore should not be used to automatically create the zpool when working with failover shared storage. This is because the restriction prevents a user from defining the correct `cachefile` property for the zpool to prevent multiple hosts from automatically importing a zpool. The `mkfs.lustre` command also will not allow a user to provide tuning options at file system create time, notably the `ashift` property that helps with write alignment for storage devices that do not correctly report the underlying sector size.

Examining ZFS Pools with **zdb**

The `zdb` (ZFS Debug) command is a useful tool for examining the low-level structure and metadata of a ZFS pool or dataset, and can read the information from the on-disk data structures of exported pools as well as from the ZFS pool cache file. The output is subject to change over time as ZFS is further developed, which means that the exact content contained in the output may vary depending on which version of ZFS is being used.

This guide will not go into the details of the `zdb` command but will highlight one or two useful features for examining the file system data structures.

To read the configuration of a zpool that is imported on a host, use the following command:

```
zdb -C[C] <poolname>
```

If the `-C` option is used, the output will display the cached content from the zpool configuration cache, as well as the on-disk configuration (called the Meta Object Set, or MOS).

For example:

```
[root@rh7z-mds1 ~]# zpool list mgspool
NAME      SIZE  ALLOC   FREE  EXPANDSZ   FRAG    CAP  DEDUP  HEALTH
ALTROOT
mgspool  1008M  2.23M  1006M         -     1%     0%  1.00x  ONLINE
-
[root@rh7z-mds1 ~]# zdb -C mgspool

MOS Configuration:
        version: 5000
        name: 'mgspool'
```

```
state: 0
txg: 34022
pool_guid: 11089712772589408485
errata: 0
hostid: 1489912803
hostname: 'rh7z-mds1'
vdev_children: 1
vdev_tree:
    type: 'root'
    id: 0
    guid: 11089712772589408485
    children[0]:
        type: 'mirror'
        id: 0
        guid: 10014780520217715169
        metaslab_array: 34
        metaslab_shift: 24
        ashift: 9
        asize: 1058537472
        is_log: 0
        create_txg: 4
        children[0]:
            type: 'disk'
            id: 0
            guid: 12166736421740210173
            path: '/dev/sda1'
            whole_disk: 1
            create_txg: 4
        children[1]:
            type: 'disk'
            id: 1
            guid: 8586593287004549671
            path: '/dev/sdc1'
            whole_disk: 1
            create_txg: 4
features_for_read:
    com.delphix:hole_birth
    com.delphix:embedded_data
```

If the pool has not been imported to the current host, or if there is no zpool cache file on the host for a currently imported pool, use the following command:

```
zdb -e <poolname>
```

This will provide a very large amount of output. To get the MOS configuration, use the following:

```
zdb -eC[C] <poolname>
```

For example:

```
[root@rh7z-mds1 ~]# zdb -eC mgspool

MOS Configuration:
        version: 5000
        name: 'mgspool'
        state: 1
        txg: 34189
        pool_guid: 11089712772589408485
        errata: 0
        hostid: 1489912803
        hostname: 'rh7z-mds1'
        vdev_children: 1
        vdev_tree:
            type: 'root'
            id: 0
            guid: 11089712772589408485
            children[0]:
                type: 'mirror'
                id: 0
                guid: 10014780520217715169
                metaslab_array: 34
                metaslab_shift: 24
                ashift: 9
                asize: 1058537472
                is_log: 0
                create_txg: 4
                children[0]:
                    type: 'disk'
                    id: 0
                    guid: 12166736421740210173
                    path: '/dev/sda1'
                    whole_disk: 1
                    create_txg: 4
                children[1]:
                    type: 'disk'
                    id: 1
                    guid: 8586593287004549671
                    path: '/dev/sdc1'
```

```
            whole_disk: 1
            create_txg: 4
features_for_read:
    com.delphix:hole_birth
    com.delphix:embedded_data
```

Note that the MOS includes the hostid of the last host to import the zpool. If the hostid of the SPL that imported the zpool was 0 (zero), then this field will not be presented in the zdb MOS output. If there is no hostid in the MOS, then this strongly indicates that the hosts are not configured with SPL hostids, and the ZFS volume will not be protected from multiple concurrent accesses from multiple hosts. Please see the chapter Protecting File System Volumes from Concurrent Access (Multi-mount Protection) for details on the importance of correctly setting the hostid for SPL when working with high availability failover storage configurations.

Optimizing Performance of SSDs and Advanced Format Drives with `zpool ashift`

The following information is taken from a FAQ entry on the ZFS on Linux project web site, reference: https://github.com/zfsonlinux/zfs/wiki/faq.

Advanced Format (AF) is a disk format that natively uses a sector size of 4,096 bytes instead of 512 bytes. To maintain compatibility with legacy systems, AF disks emulate a sector size of 512 bytes.

The default behavior of ZFS is to automatically detect the sector size of the drive. However, when attempting to detect the sector size of an AF drive, ZFS will not be able to detect the native sector size, and will instead trust the drive when it reports the emulated size. This can result in poorly aligned disk access that degrades performance of the pool.

Therefore, the ability to set the `ashift` property has been added to the `zpool` command. This allows users to explicitly assign the sector size when devices are first added to a pool (typically at pool creation time or when adding a `vdev` to the pool). The `ashift` values range from 9 to 16, with the default value 0 meaning that ZFS should auto-detect the sector size.

This issue is rare on hard disk drives, and so it might not be necessary to alter the ashift value. However, there are reports that SSDs can benefit from setting the ashift property explicitly to match the 4096 byte sector size.

The value of `ashift` is actually a bit shift value, so the `ashift` value for 512 bytes is 9 (2^9 = 512) while the `ashift` value for 4,096 bytes is 12 (2^{12} = 4,096). To force the pool to use 4,096 byte sectors at pool creation time:

```
$ sudo zpool create -o ashift=12 tank mirror sda sdb
```

To force the pool to use 4,096 byte sectors when adding a `vdev` to a pool:

```
$ sudo zpool add -o ashift=12 tank mirror sdc sdd
```

ZFS `recordsize` Property

The `recordsize` property of ZFS datasets is used to specify the maximum block size for files in the file system. Normally this property should not be changed, but for workloads that create very large files, increasing the value of `recordsize` can deliver a performance benefit. The chosen size must be a power of 2 with the minimum allowed size being 512 bytes. For ZFS on Linux, the maximum value is 1MiB in version 0.6.5. Prior to this release, the maximum value was 128K (which is also the default setting).

The default setting is not sufficient to sustain the performance of the throughput oriented workloads that are typical of the I/O patterns for OSTs, and it is recommended to increase the `recordsize` to 1MiB (1024K), in order to better match the Lustre IO transaction sizes for block I/O.

Protecting File System Volumes from Concurrent Access

Storage volumes formatted for ldiskfs (itself based on EXT4) and ZFS are not SAN-aware parallel file systems and do not support multiple concurrent accesses from different computers. If two or more computers attempt to write directly to the same storage, the write operations will not be coordinated, which could lead to data corruption: each host that has the storage mounted will assume that they have unique access to the data and will not take into account any IO transactions external to that host.

For this reason, Lustre OSDs must be mounted by no more than one host at any single point in time. This also has the consequence when working with high availability frameworks that each individual storage target can only participate in an HA cluster as a failover resource (also referred to as an active-passive resource).

Administrators must take this into consideration when planning Lustre file system deployments and when conducting any maintenance.

To protect against data corruption, the ldiskfs OSD format has built-in protection against multiple concurrent mounts of a storage volume, which is referred to as “multi-mount protection” (MMP). The storage will refuse to mount if the host detects that the storage might already be mounted elsewhere.

OpenZFS also provides some limited protection against concurrent mounts, although this is a superficial mechanism that can be circumvented if one is not careful. OpenZFS is not a multi-host aware file system, so it relies on external controls, such as the resource agents in a Pacemaker framework, to enforce isolation of zpools to a single host when running in an HA cluster. In OpenZFS on Linux, the Solaris Portability Layer (SPL) relies on a system setting called the `hostid`, which is intended to uniquely identify a computer. To help reduce the risk

of a zpool against being imported by many hosts concurrently, the hostid of the system where the zpool was created, or where the zpool was last successfully imported, is written into the pool configuration. The zpool therefore has a notion of system ownership written into its configuration. If an attempt is made to import a zpool that has the hostid set to a value that does not match the hostid of the system where the import is being executed, the attempt fails. Exporting the zpool will clear the hostid.

The import can be forced, which will override the hostid check. When forcing the import of a ZFS pool, be very careful to ensure that the volume is not currently imported anywhere else.

Unfortunately, the SPL, by default, sets the value of its internal hostid to 0 (zero), based on the result of the `gethostid()` system call, and this is what is written into the zpool. This means that, without any additional configuration of the operating system after installation of the ZFS and SPL software, the SPL hostid will be 0 across all hosts, regardless of the actual system hostid. When a zpool is created and imported, it will inherit this hostid value of zero and write that into the zpool configuration. Now, any system with a hostid of 0 will be able to import the zpool because the zpool import command will always succeed when the hostid of the system matches the hostid of the zpool.

Therefore, by relying on the default configuration, the check against concurrent access by multiple hosts is lost, leading to a corruption of the zpool and data loss. It cannot be emphasized strongly enough that ZFS pools should never, ever be imported to two nodes at the same time, as serious corruption will occur.

It is easy to be fooled by the operating system runtime that the hostid is configured, because the `hostid` command in the GNU `coreutils` package will return a non-zero unique value. Example output:

```
[root@rh7z-mds1 ~]# hostid
460a0be3
```

Unfortunately, the SPL cannot make use of this utility to derive the hostid of a system. To determine the SPL's active hostid value, it must be retrieved from `/sys/module/spl/parameters/spl_hostid` in the kernel sysfs interface. In the following example, the SPL hostid is 0 (zero):

```
[root@rh7z-mds1 ~]# cat /sys/module/spl/parameters/spl_hostid
0
```

Each host with ZFS storage must have a unique but persistent (i.e., unchanging) hostid that can be used by the SPL. SPL can be provided with a hostid via the kernel command line on system boot, or by using the `genhostid` command to create a random hostid which is written to the file `/etc/hostid`. Note that the `genhostid` writes its output in binary to the file, so it is not readable without translation by a utility such as `hexdump`, `od` or `xxd`.

The `genhostid` command is the easiest option to implement and is recommended for setting the hostid for SPL. Run the command as follows (the command does not generate any output to the shell):

```
[root@rh7z-mds1 ~]# genhostid
```

Verify that data has been written to the file. This example uses xxd to convert the file content to hexadecimal:

```
[root@rh7z-mds1 ~]# xxd -p /etc/hostid
e343ce58
```

The `genhostid` command must be run on every Lustre server that has or will have ZFS volumes configured. Make sure that each server gets a unique hostid. Reboot each system after the hostid is configured in order for the SPL kernel module to pick up the change.

The other essential step in protecting a ZFS pool from concurrent mounts across hosts is to prevent the pool from being automatically imported on system boot. This is controlled by the content of the system default ZFS pool configuration cache, or cache file, usually written to `/etc/zfs/zpool.cache`.

The configuration cache keeps a record of the configuration of each zpool that is either created or imported to a host. If a zpool is exported, it is removed from the configuration cache. Note that any exported zpools that are detected by the host on system boot will be automatically [re-]imported and added back into the default configuration cache.

The cache file is a generally useful feature that can speed up the "assembly" of zpools on system boot, but special care must be taken when managed pools that are kept on shared storage and participate in high availability server configurations.

Caution: Any zpool that has its configuration recorded in the default cache file will be imported by the host on system boot automatically, *even if that pool has already been imported on a different host*. The `zpool` command effectively assumes that the content of a cache file is accurate and reliable, and so any pool configurations stored in the cache are valid for importation without further checks. *This is very dangerous for ZFS storage that is shared across multiple hosts.*

The behavior of the zpool command in an HA environment can be observed with the following experiment.

Caution: Do not try this with any ZFS storage containing production data, as the experiment will very likely lead to data corruption:

1. Connect two hosts to a shared storage enclosure and ensure that each host can access the devices in the enclosure.

2. Install ZFS onto each server and configure the hostid for each server as well. Reboot to ensure all changes have been applied.
3. On the first host (A), create a zpool with the default cache file:

   ```
   zpool create tank mirror sda sdb
   ```

4. Power down host A
5. On the second host (B), attempt to import the zpool without force:

   ```
   zpool import tank
   ```

6. The command will fail and an error similar to the following will be produced:

   ```
   [root@rh7z-mds2 system]# zpool import tank
   cannot import tank: pool may be in use from other system, it was
   last accessed by rh7z-mds1 (hostid: 0x58ce43e3) on Wed Mar 23
   02:04:45 2016
   use '-f' to import anyway
   ```

7. Force the import of the pool:

   ```
   zpool import -f tank
   ```

8. Now the zpool will be imported onto host B.
9. Power on host A.
10. During system boot, the following command will be run by the `zfs-import-cache` service:

    ```
    /sbin/zpool import -c /etc/zfs/zpool.cache –aN
    ```

    The effect of this command is to automatically import all zpools listed in the system default cache file, `/etc/zfs/zpool.cache`. Because the zpool was not exported from host A before it was powered off, the cache file still contains the configuration information for the zpool, which is why it is automatically imported.
11. When host A has completed the boot process, the zpool that was created will now be simultaneously imported on both host A and B. This puts the data held within the pool at risk of corruption.

To prevent a zpool from being imported automatically on boot, use the `cachefile` option to create a separate, unique cache file for the pool being created, or set `cachefile=none`. This is only effective when the hostid has also been set correctly for SPL, as described earlier.

Note that the `cachefile` option has to be specified every time the `zpool import` command is run, not just when creating the pool or running the import command for the first

time. It must also be specified when running the `import` command on a failover host in an HA cluster. Also be aware that the value `none` is a reserved keyword and should not be used as a file name. Setting the `cachefile` parameter equal to the empty string (`''` or `""`) is the same as telling ZFS to use the default cache file.

For high-availability clusters with ZFS storage shared between nodes as a failover resource, it is recommended that each zpool is created and imported with the `cachefile` set equal to the special value of `none`:

```
zpool create [-f] -O canmount=off -o cachefile=none \
  [-o ashift=<n>] \
  <zpool name> <zpool specification>

zpool import [-f] -o cachefile=none <zpool name>
```

If the hostid for the servers are set properly, and the cachefile property is set to none, then the system boot services will not force an import of a zpool that has been imported on a different server. However, for further safety, one can also disable the ZFS storage services from attempting to automatically start on system boot. This will mean that the host will not attempt any automated import of ZFS storage, which might prevent a double import of a pool onto multiple hosts.

Note that if there are any non-Lustre storage devices formatted using ZFS, they will also be affected by this change and will not be available until explicitly imported after system start-up.

The simplest way to disable the services is to disable the ZFS target milestone:

```
systemctl disable zfs.target
```

Using ZFS Properties to Protect Lustre OSDs

To protect the integrity of ZFS volumes used by Lustre, the `zpool` command should be invoked with an option to set the property, `canmount=off`, when working with Lustre storage volumes. This property will also be automatically applied to any ZFS datasets created by the `mkfs.lustre` command.

The property `canmount=off` is used to prevent a dataset within a pool from being mounted by the standard ZFS tools, e.g., by executing the `zfs mount -a` command, thus preventing accidental and incorrect mounts of ZFS storage that is being used for Lustre. Setting the property in the zpool command ensures that all of the datasets in the zpool inherit this property. It will also ensure that the file system datasets that have been formatted for use by Lustre will not get mounted on system boot by the ZFS services in `systemd` or `sysvinit` (on hosts running RHEL 7, for example, the `systemd zfs-mount` service will run "`zfs mount -a`" during system startup).

However, note that the `canmount` parameter will not prevent a zpool from being *imported* by a host on system boot, it will only stop the datasets in an imported zpool from being mounted. If ZFS detects a zpool on system startup that is eligible for import, it will do so automatically. Users must be very careful when managing ZFS volumes to ensure that the zpools are only imported onto a single host at a time.

The `zfs` command-line executed by `mkfs.lustre` also sets the `xattrs=sa` property. This is used to improve performance of the ZFS storage, especially when using POSIX ACLs (Access Control Lists). SA stands for System Attributes, and provides an alternative implementation to the default Directory-based extended attributes. Storing extended attributes using system attributes significantly decreases disk I/O and is recommended for systems that make use of SELinux or POSIX ACLs. Refer to the ZFS manual page for a more detailed explanation.

Many of the ZFS properties can be altered after a zpool or dataset has been created, and formatting a Lustre target using the ZFS OSD will always set the ZFS properties `canmount=off` and `xattrs=sa`.

Create the Management Service (MGS)

The Management Service, or MGS, is the most straightforward Lustre file system service to create. The MGS is a global resource that can support multiple file systems in a service domain.

The MGS has very resource-modest requirements compared to the Metadata Service (MDS) and the Object Storage Service (OSS), as it is not a compute- or storage-intensive service. The storage required for the MGT is somewhere in the region of 100-200MiB, and any physical storage allocation is likely to far exceed this minimum requirement. Therefore the minimum real-world requirement is a fault-tolerant storage device, ideally a two-disk, RAID 1 mirror. Some storage arrays support the creation of a small virtual disk from a larger physical array configuration. For example, one might create a physical storage array consisting of 24 disks in a RAID 10 configuration, and split this into 2 vdisks: a small 1GB vdisk for the MGT, with the remainder allocated to e.g. MDT0. Each vdisk is presented as an independent block device to the servers, and can be managed independently from the point of view of failover configuration.

The MGS can run on a standalone server, but like all Lustre services, if that host fails, the MGS will be offline until the host can be restored. Therefore, the MGS is most often deployed into an HA failover configuration, with a small shared storage device or volume that can be mounted on more than one host. Because the MGS consumes only a small amount of a server's resources, it is unusual to create an HA cluster that contains only the MGS. Instead, the MGS will typically be paired with the root MDS (i.e., the metadata service for MDT0, which is the storage target that contains the root of a Lustre file system namespace).

For truly flexible high -availability configurations, where resources are somewhat autonomous and can be managed independently, the MGT storage should be allocated to a device or volume that is independent of the MDT storage, from the perspective of the OS (although both MGT and MDT might be in the same physical storage enclosure). The intention is that each service can migrate independently between hosts in a high availability server configuration.

For ldiskfs-based storage, this generally means that the MGT and MDT are contained on separate LUNs or vdisks in a storage array, and for ZFS, the MGT and MDT should be in separate zpools (when using ZFS for the MGT and MDT storage in a high availability configuration, do not configure the MGT and MDT as datasets in the same pool. The pool can only be imported onto one host at a time, which will prevent the services from running on separate hosts, and will not allow independent service migration or failover).

There can only be one MGS running on a node at one time. This means that one cannot have multiple MGTs configured in the same HA cluster, because even if the services initially start on separate nodes, if a failover occurs, they will both end up being located on the same host.

Incoming client connections will not be able to determine which service to connect to, and may be connected arbitrarily, resulting in the registration and other configuration data being split in unpredictable ways across the competing resources, with the likely effect of corrupting the configuration on both targets. There is no specific requirement in Lustre to create multiple MGS; one MGS will suffice for many file systems in a subnet.

MGT Formatted as a ZFS OSD

Formatting the MGT using only the **`mkfs.lustre`** command

The syntax for creating a ZFS-based MGT using only the `mkfs.lustre` command is as follows:

```
mkfs.lustre --mgs \
  [ --reformat ] \
  [ --servicenode <NID> [--servicenode <NID> …]] \
  [ --failnode <NID> [--failnode <NID> …]] \
  --backfstype=zfs \
  [ --mkfsoptions <options> ] \
  <zpool>/<dataset> <zpool specification>
```

This example uses the `--servicenode` syntax to create an MGT that can be run on two servers as a high availability failover resource:

```
[root@rh7z-mds1 ~]# mkfs.lustre --mgs \
  --servicenode 192.168.227.11@tcp1 \
  --servicenode 192.168.227.12@tcp1 \
  --backfstype=zfs \
  mgspool/mgt mirror sda sdc
```

The command-line formats a new MGT that will be used by the MGS for storage. The command further defines a mirrored zpool called `mgspool` (consisting of two devices) and creates a ZFS dataset called `mgt`. Two server NIDs are supplied as service nodes for the MGS, `192.168.227.11@tcp1` and `192.168.227.12@tcp1`.

The failnode syntax is similar, but is used to define only a failover target for the storage service. For example:

```
[root@rh7z-mds1 ~]# mkfs.lustre --mgs \
  --failnode 192.168.227.12@tcp1 \
  --backfstype=zfs \
  mgspool/mgt mirror sda sdc
```

Here, the failover host is identified as `192.168.227.12@tcp1`, one server in an HA pair (and which, incidentally, has the hostname `rh7z-mds2`). The `mkfs.lustre` command was executed on `rh7z-mds1` (NID: `192.168.227.11@tcp1`), and the `mount` command must also be run from this host when the MGS service starts for the very first time.

Formatting the MGT using `zpool` and `mkfs.lustre`

To create a ZFS-based MGT, create a zpool to contain the MGT file system dataset, then use `mkfs.lustre` to actually create the file system inside the zpool:

```
zpool create [-f] -O canmount=off \
  [-o ashift=<n>] \
  -o cachefile=/etc/zfs/<zpool name>.spec | -o cachefile=none \
  <zpool name> <zpool specification>

mkfs.lustre --mgs \
  [ --reformat ] \
  [ --servicenode <NID> [--servicenode <NID> …]] \
  [ --failnode <NID> [--failnode <NID> …]] \
  --backfstype=zfs \
  [ --mkfsoptions <options> ] \
  <pool name>/<dataset>
```

For example:

```
# Create the zpool
zpool create -O canmount=off \
  -o cachefile=none \
  mgspool mirror sda sdc

# Format the Lustre MGT
mkfs.lustre --mgs \
  --servicenode 192.168.227.11@tcp1 \
  --servicenode 192.168.227.12@tcp1 \
  --backfstype=zfs \
  mgspool/mgt
```

Use the `zfs get` command to retrieve comprehensive metadata information about the file system dataset and to confirm that the Lustre properties have been set correctly:

```
zfs get all | awk '$2 ~ /lustre/'
```

Alternatively, use the following command to retrieve only those properties that have been explicitly set:

```
zfs get all -s local
```

An MGT example:

```
[root@rh7z-mds1 ~]# zfs get all -s local
NAME         PROPERTY              VALUE                                  SOURCE
mgspool      canmount              off                                    local
mgspool/mgt  canmount              off                                    local
mgspool/mgt  xattr                 sa                                     local
mgspool/mgt  lustre:version        1                                      local
mgspool/mgt  lustre:index          65535                                  local
mgspool/mgt  lustre:failover.node  192.168.227.11@tcp1:192.168.227.12@tcp1  local
mgspool/mgt  lustre:svname         MGS                                    local
mgspool/mgt  lustre:flags          4132                                   local
```

Starting and stopping the MGS Service

The mount command is used to start all Lustre storage services, including the MGS. Therefore, to start up the MGS, one must mount the MGT on the server. The syntax is:

```
mount -t lustre [-o <options>] \
  <ldiskfs blockdev>|<zpool>/<dataset> <mount point>
```

The mount command syntax is very similar for both LDISKFS and ZFS MGT storage targets. The main difference is the format of the path to the storage. For ldiskfs, the path will resolve to a block device, such as `/dev/sda` or `/dev/mapper/mpatha`, whereas for ZFS, the path resolves to a dataset in a zpool, e.g., `mgspool/mgt`.

The following example starts a ZFS-based MGT:

```
# Ignore MOUNTPOINT column in output: not used by Lustre
[root@rh7z-mds1 ~]# zfs list
NAME          USED  AVAIL  REFER  MOUNTPOINT
mgspool      1.67M   974M    19K  /mgspool
mgspool/mgt  1.59M   974M  1.59M  /mgspool/mgt

[root@rh7z-mds1 ~]# mkdir -p /lustre/mgt

[root@rh7z-mds1 ~]# mount -t lustre mgspool/mgt /lustre/mgt

[root@rh7z-mds1 ~]# df -ht lustre
File system      Size  Used Avail Use% Mounted on
mgspool/mgt     960M  1.7M  957M   1% /lustre/mgt
```

Note that the default output for `zfs list` shows the mount points for the MGS pool and MGT dataset in the `MOUNTPOINT` column. The columns presented in the output can be changed to simplify presentation of the table. For example:

```
[root@rh7z-mds1 ~]# zfs list -o name,used,avail,refer
NAME          USED  AVAIL  REFER
mgspool      2.20M   974M    19K
mgspool/mgt  2.09M   974M  2.09M
```

The content in the `MOUNTPOINT` column of the default output can be ignored, as the referenced mount points are not used for mounting Lustre ZFS OSDs. Instead, create a mount point explicitly, just as for an LDISKFS-based storage target.

The recommended convention for the mount point of the MGT storage is `/lustre/mgt`.

Remember that only the `mount -t lustre` command can start Lustre services. Mounting storage as type `ldiskfs` or `zfs` will mount a storage target on the host, but it will not trigger the startup of the requisite kernel processes.

To verify that the MGS is running, check that the device has been mounted, then get the Lustre device list with `lctl dl` and review the running processes:

```
[root@rh7z-mds1 lustre]# df -ht lustre
File system      Size  Used Avail Use% Mounted on
mgspool/mgt     960M  2.0M  956M   1% /lustre/mgt

[root@rh7z-mds1 ~]# lctl dl
  0 UP osd-zfs MGS-osd MGS-osd_UUID 5
  1 UP mgs MGS MGS 5
  2 UP mgc MGC192.168.227.11@tcp1 5d62a612-f872-09a4-7da8-
4ce562af6e0c 5

[root@rh7z-mds1 ~]# ps -ef | awk '/mgs/ && !/awk/'
root     15162     2  0 02:44 ?        00:00:00 [mgs_params_noti]
root     15163     2  0 02:44 ?        00:00:00 [ll_mgs_0000]
root     15164     2  0 02:44 ?        00:00:00 [ll_mgs_0001]
root     15165     2  0 02:44 ?        00:00:00 [ll_mgs_0002]
```

To stop a Lustre service, umount the corresponding target:

```
umount <mount point>
```

The mount point must correspond to the mount point used with the `mount -t lustre` command. For example:

```
[root@rh7z-mds1 ~]# df -ht lustre
File system      Size  Used Avail Use% Mounted on
mgspool/mgt     960M  2.0M  956M   1% /lustre/mgt
[root@rh7z-mds1 ~]# umount /lustre/mgt
```

```
[root@rh7z-mds1 ~]# df -ht lustre
df: no file systems processed
[root@rh7z-mds1 ~]# lctl dl
[root@rh7z-mds1 ~]#
```

Using the regular `umount` command is the correct way to stop a given Lustre service and unmount the associated storage, both ldiskfs and ZFS-based Lustre storage volumes.

Do not use the `zfs unmount` command to stop a Lustre service. Attempting to use `zfs` commands to unmount a storage target that is mounted as part of an active Lustre service will return an error:

```
[root@rh7z-mds1 ~]# lctl dl
  0 UP osd-zfs MGS-osd MGS-osd_UUID 5
  1 UP mgs MGS MGS 5
  2 UP mgc MGC192.168.227.11@tcp1 be9fad27-107b-d165-8494-
9a723b90e863 5

[root@rh7z-mds1 ~]# mount -t lustre
mgspool/mgt on /lustre/mgt type lustre (ro)

[root@rh7z-mds1 ~]# zfs list
NAME           USED  AVAIL  REFER  MOUNTPOINT
mgspool       2.05M   974M    19K  /mgspool
mgspool/mgt   1.97M   974M  1.97M  /mgspool/mgt

[root@rh7z-mds1 ~]# zpool status
pool: mgspool
 state: ONLINE
  scan: none requested
config:

   NAME        STATE     READ WRITE CKSUM
   mgspool     ONLINE       0     0     0
     mirror-0  ONLINE       0     0     0
       sda     ONLINE       0     0     0
       sdc     ONLINE       0     0     0

errors: No known data errors

[root@rh7z-mds1 ~]# zfs unmount mgspool/mgt
cannot unmount 'mgspool/mgt': not currently mounted

[root@rh7z-mds1 ~]# zfs unmount /lustre/mgt
cannot unmount '/lustre/mgt': not a ZFS file system
```

In the example, the MGS is up and running on a host, and the MGT storage is formatted as a ZFS dataset in a mirrored zpool. The service is online and the storage is mounted as a Lustre file system type. When an attempt is made to use ZFS to umount the volume, the command fails, regardless of if one uses `<zpool>/<dataset>` or the mount point as the reference to the storage volume.

These examples are provided to reinforce the point that many of the Lustre server management tools are the same whether ldiskfs or ZFS is used for the underlying storage. Of course there are storage-level differences, but where possible, the Lustre tools are common to both storage target formats.

Create the Metadata Service (MDS)

The Metadata Service, or MDS, provides the index, or namespace, service for a Lustre file system. The metadata content is stored on object storage device (OSD) volumes called Metadata Targets (MDTs); a Lustre file system's namespace (the directory structure and file names), permissions, extended attributes, and file layouts are recorded to the MDTs. Each Lustre file system must have a minimum of one MDT, called MDT0, which contains the root of the file system namespace, but a single Lustre file system can have many MDTs, providing a scalable file system index for very large-scale name spaces with complex directory structures and very large quantities of files.

The metadata service also controls the layout of files (number of stripes and stripe size), and is responsible for object allocation on the OSTs. The MDS determines and allocates a file's layout, but an application or user on a Lustre client can override the default layout when a file is created. It is possible to create layouts for each file in the file system in order to optimize the IO for a given workload. For convenience, a default policy for the file layout can be assigned to a directory so that each file created within that directory will inherit the same layout format. Files do not share OST objects, so each individual file has a unique storage layout, although many files will conform to the same layout policy.

The MDS is only involved in metadata operations for a file or directory. After a file has been opened, the MDT does not participate in I/O transactions again until the file is closed, avoiding any overheads that might be incurred by an application switching between throughput and metadata work.

Prior to Lustre version 2.4, only a single MDT could be used to store the metadata for a Lustre file system. With the introduction of the Distributed Name Space (DNE) feature, the metadata workload for a single file system can be distributed across multiple MDTs, and consequently multiple metadata servers. There are two implementations of distributed metadata available to file system architects: remote directories, and striped directories.

DNE remote directories – sometimes referred to as DNE phase 1 or DNE1 – provide a way for administrators to assign a discrete sub-tree of the overall file system namespace to a specific MDT. In this way, the MDTs are connected into a virtual tree structure, with each MDT associated with a specific sub-directory. MDT0 is always used to represent the root of the name space, with all other MDTs as a child of MDT0.

While it is technically possible to create nested MDT relationships, this is disabled by default and discouraged as an architecture, because loss of an MDT means loss of access to the subdirectories hosted by that MDT and by extension, any content on MDTs that were serving subdirectories more deeply nested in the tree than the failed MDT.

DNE striped directories – sometimes called DNE phase 2 or DNE2 – uses a more sophisticated structure to load balance metadata operations across multiple servers and MDTs. With striped

directories, the structure of a directory is split into shards that are distributed across multiple MDTs. The user defines the layout for a given directory. Directory entries are split across the MDTs and the inodes for a file are created on the same MDT that holds the file name entry.

A single metadata server can serve the content of many MDTs, even MDTs for different file systems. When metadata servers are grouped into high-availability cluster configurations, this capability allows MDT resources to be configured to run on each host in the cluster configuration, so that each server can be actively providing service to the network, with no idle "standby" nodes. When a server fails or requires maintenance, its MDT resources can migrate to other nodes without conflict, preserving operational continuity of the metadata services.

Hosting the MDTs for multiple file systems on a common set of servers is not unusual, especially where there are budgetary or physical constraints, such as limited space or power in a data center, or where there is a desire for multiple file systems to be established in a single environment. This MDT hosting method provides a way to maximize utilization of the available hardware infrastructure. As with any arrangement where resources are shared between multiple services, care must be taken to ensure balance is maintained between the competing processes.

The metadata service can be configured for high availability, as can all of the services of a Lustre file system. The most common high-availability metadata server design pattern is a two-node configuration that comprises an MGS and the MDS for MDT0:

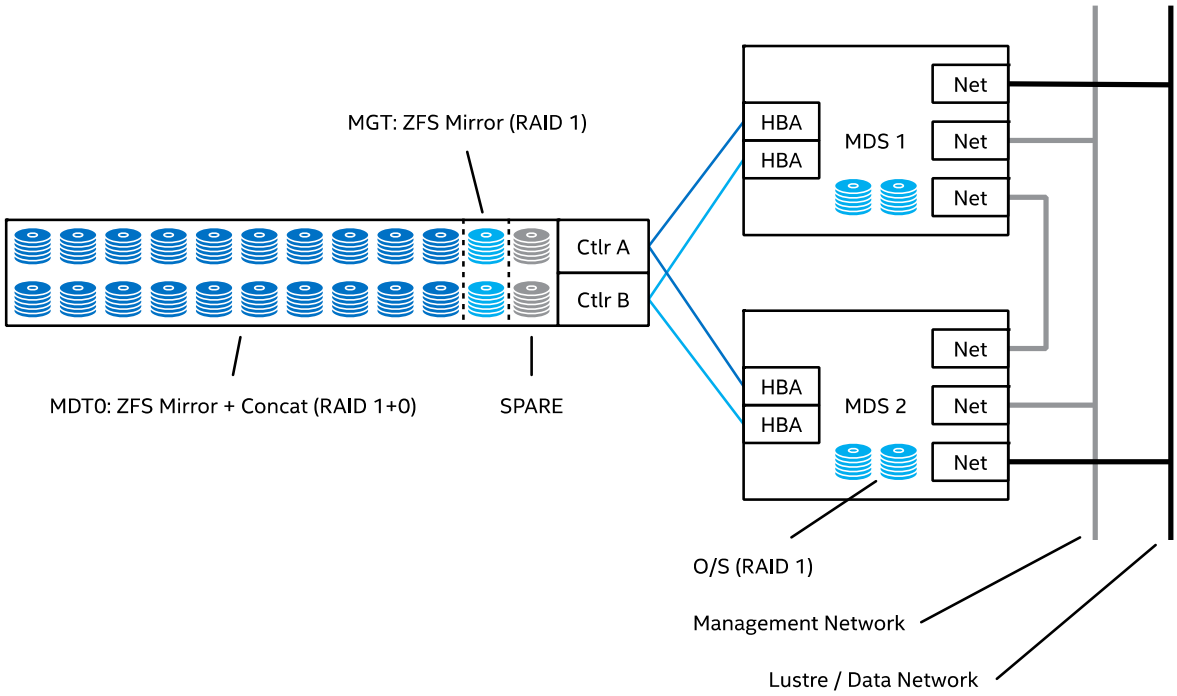

Figure 6. Typical Two-Node Metadata Server Cluster

In this arrangement, two servers are connected to an external storage enclosure that has been configured with two storage volumes: MDT0 for the Metadata service and one MGT for the Management Service.

Because the node that normally just runs the MGS is underutilized on its own, there is sufficient available capacity to run an addition MDS on the same host, provided care is taken to ensure that the MGS and additional MDS resources can be operated and migrated independently of one another within a cluster framework. This can be exploited to enable the MDT of a second file system to be hosted within the same HA cluster configuration, or to provide an additional MDT to an existing file system.

Figure 7 shows an example configuration where there are 2 MDT OSDs, in separate storage enclosures, along with the MGT, which in this case has been created by mirroring 2 drives, one from each enclosure. This maximizes the available storage for metadata bandwidth while still leaving room for a spare drive in each enclosure.

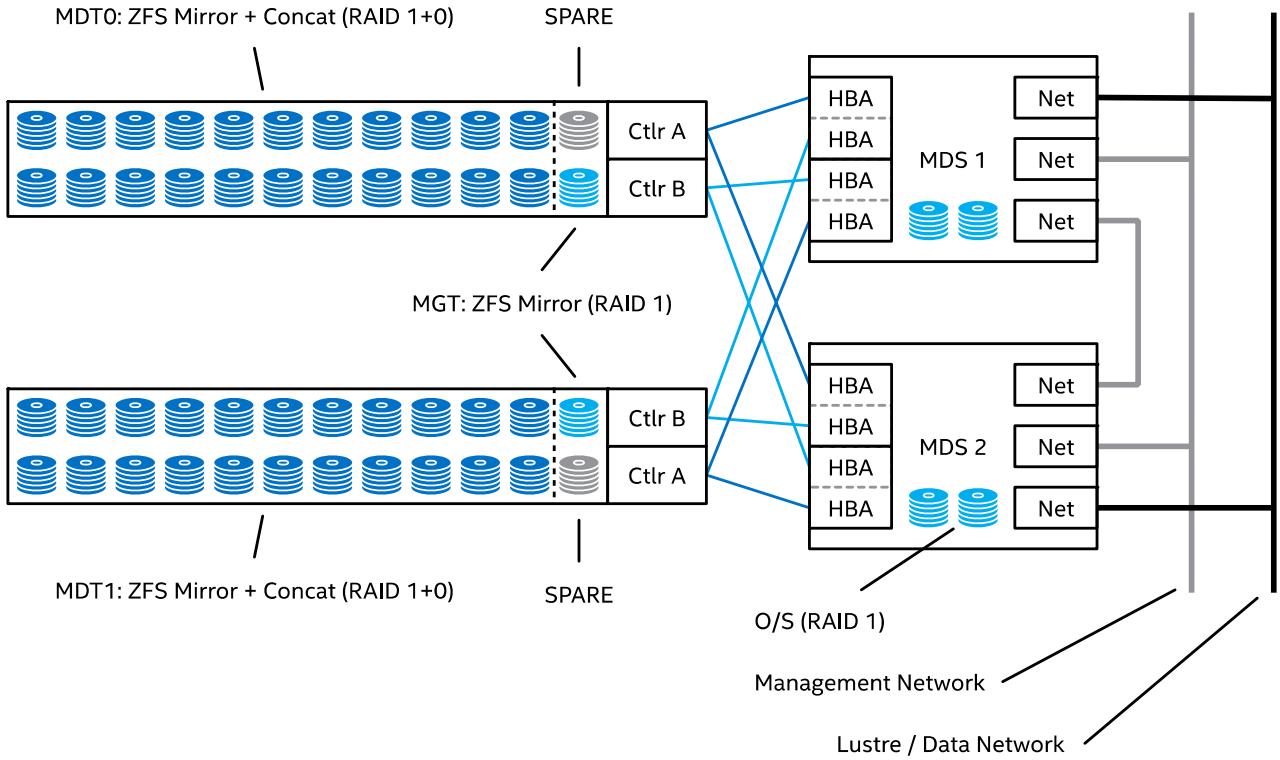

Figure 7. Metadata Server Cluster with MGS and Two MDTs

The Metadata Service is very resource-intensive and systems will benefit from frequency-optimized CPUs: clock speed and cache are more important than the number of CPU cores, although when there are a large number of service threads running, spreading the work across cores will also derive a benefit. The MDS will also exploit system memory to good advantage as cache for metadata and for lock management. The more RAM a metadata server can access, the better able it is to deliver strong performance for the concurrent workloads typical in HPC

environments. On ZFS-based platforms, system memory is also used extensively for caching, providing significant improvements in application performance.

Services can scale from very small deployments for testing, such as VMs with perhaps 2 VCPU cores and as little as 2GB or RAM, up to high-end production servers with 24 cores, 3 GHz CPUs and 512GB RAM. Sizing depends on anticipated workload and file system working set population, and as computing requirements evolve, requirements invariably become more demanding. When a metadata target is formatted, the number of inodes for the MDT is calculated using the ratio of 2KB-per-inode. The inode itself consumes 512 bytes, but additional blocks can be allocated to store extended attributes when the initial space is consumed. This happens for example, when a file has a lot of stripes. Using this 2KB-per-inode ratio has proven to be a reliable way to allocate capacity for the MDT.

At the time of writing, a typical configuration for a metadata server is to allocate 16 cores and 256GB RAM. If it is assumed that to cache a single inode in RAM without a lock requires approximately 2KB, this allows somewhere in the region of 130 million inodes to be cached in memory, depending on the other operating system overheads, which would represent around 13% of a 1 billion inode name space. A subset of these files will also be part of an active working set, which incurs an additional overhead for locking. Every client that accesses a file will require a lock, which is around 1KB per file per client. For example, if a thousand clients open a thousand files, this would incur an additional 1GiB of RAM for the locks. When sizing memory requirements, aiming for an active working set in cache of between 10 - 20 per cent of the total file system name space is a reasonable goal..

File systems with large active working sets may require an increase in RAM to achieve optimal performance, and metadata-intensive workloads with high IOps activity will benefit from additional CPU power, with the largest benefit being derived from higher clock speeds rather than very large core counts.

Metadata storage is subjected to small, random I/O, that is very IOps-intensive and somewhat transactional in nature, and MDT I/O bears many of the same attributes common for an OLTP database. A metadata server with a large active dataset can process tens of thousands to hundreds of thousands of very small I/O operations per second. High-speed storage is therefore essential and SAS storage is commonly used, but there is an increase in the use of flash storage for metadata.

Storage should be arranged in a RAID 1+0 (striped mirrors) layout to provide the best balance between performance and reliability, without the overheads introduced by RAID 5/6 or equivalent parity-based layouts. Metadata workloads are small, random IO operations, and will perform poorly on RAID 5/6 layouts because the IO is very unlikely to fit neatly into a single full stripe write, thus leading to read-modify-write overheads when recalculating parity to write data to the storage. With ZFS, the IOps are spread across the virtual devices (vdevs) in a zpool, which effectively means that the more datasets in the pool, the better the overall

performance. A pool containing many mirrored vdevs will provide better IOps performance than a pool with a single vdev.

When formatting MDT storage, Lustre will assume a ratio of 2KB per inode and allocate the on-disk format accordingly (note that the ratio of inodes to storage capacity is not the same as the size of the inode itself, which is 512 bytes on LDISKFS storage. Additional space is used on the MDT for extended attributes to contain the file layout and other information, which is one of the reasons why this ratio is chosen). Metadata storage formatted using ldiskfs cannot allocate more than 4 billion inodes, a limitation in the EXT4 file system upon which ldiskfs is based. Because of the 2KB/inode allocation ratio, this puts the maximum volume size for an ldiskfs MDT at 8TB. Attempting to format a storage volume for ldiskfs that is larger than 8TB will fail. Even if the format were to succeed, capacity would be wasted. ZFS storage does not suffer from this limitation.

When considering OSD formats for the MDT, be aware that ldiskfs will out-perform ZFS for metadata-intensive workloads. This is in part due to the additional data-integrity checks and data-protection overheads that are part of an IO transaction in ZFS: checksums calculated to protect against data corruption must also be written out to the ZFS storage, and the volume manager has to ensure that blocks get duplicated across mirrors (or parity calculations generated and blocks written out for RAIDZ{1,2,3} storage layouts). Note that in ZFS, checksums are stored in the block pointer to a block, not the block itself; when the checksum is written, the checksum of the block pointer must also be updated, and so on up the file system tree. Data integrity assurance and protection from corruption are not free.

Also consider allocating additional storage capacity for recording snapshots. Snapshots of MDTs can be very useful for providing a means to create online backups of the metadata for a file system, without incurring an outage of the file system. Catastrophic loss of the MDT means loss of the file system name space, and consequently loss of the index to the data objects for each file that are held on the Object Storage Servers. In short, loss of the MDT renders a Lustre file system unusable. If, however, a regular backup is made of the metadata, then it is possible to recover a file system back to production state. Using a snapshot makes it easier to take a copy of the MDT while the file system is still online. ZFS snapshots are very efficient and introduce very little overhead. Ldiskfs does not have any built-in snapshot capability, but it is possible to use LVM to create a logical volume formatted for ldiskfs and use LVM to create snapshots. Be aware that LVM snapshots can degrade the performance of storage significantly, so snapshots should be destroyed after the backup has been successfully completed.

It is technically possible to combine the MGT and MDT into a single LUN, however this is strongly discouraged. It reduces the flexibility of both services, makes maintenance more complex, and does not allow for distribution of the services across nodes in an HA cluster to optimize performance. When designing Lustre high availability storage solutions, do not combine the MGT and MDT into a single storage volume.

MDT Formatted as a ZFS OSD

In common with all Lustre ZFS object storage devices, the process for formatting a new MDT file system target can be encapsulated entirely within a single invocation of the `mkfs.lustre` command, unless high availability is required (i.e., where the file system target is held on devices in a shared storage enclosure, which is connected to 2 or more hosts).

When high availability is required, the creation of the zpool for the MDT must be separate from formatting the ZFS dataset as a Lustre storage target.

Formatting an MDT using only the **`mkfs.lustre`** command

The syntax for creating a ZFS-based MDT using only the `mkfs.lustre` command is as follows:

```
mkfs.lustre --mdt \
  [--reformat] \
  --fsname <name> \
  --index <n> \
  --mgsnode <MGS NID> [--mgsnode <MGS NID> …] \
  [ --servicenode <NID> [--servicenode <NID> …]] \
  [ --failnode <NID> [--failnode <NID> …]] \
  --backfstype=zfs \
  [ --mkfsoptions <options> ] \
  <pool name>/<dataset> \
  <zpool specification>
```

The command line syntax for formatting an MDT incorporates several additional parameters when compared to that of the much simpler MGT. First, the metadata target requires a record of the NIDs that can provide the Lustre management service (MGS). The MGS NIDs are supplied using the `--mgsnode` flag. If there is more than one potential location for the MGS (i.e., it is part of a high availability failover cluster configuration), then the option is repeated for as many failover nodes as are configured (usually there are two). Ordering is significant: the first `--mgsnode` flag must reference the NID of the current active or primary MGS server. If this is not the case, then the first time that the MDS tries to join the cluster, it will fail. The first time mount of a storage target does not currently check the failover targets. When adding new storage targets to Lustre, the MGS must be running on its primary NID.

The MDT must also be supplied with the name of the Lustre file system (maximum 8 characters), and a unique index number for the file system. There must always be an MDT with index=0 (zero) for each file system, representing the root of the file system tree. For the majority of Lustre file systems, a single MDT (referred to as MDT0) is sufficient.

This example uses the `--servicenode` syntax to create an MDT that can be run on two servers as an HA failover resource:

```
[root@rh7z-mds2 system]# mkfs.lustre --mdt \
>   --fsname demo \
>   --index 0 \
>   --mgsnode 192.168.227.11@tcp1 --mgsnode 192.168.227.12@tcp1 \
>   --servicenode 192.168.227.12@tcp1 \
>   --servicenode 192.168.227.11@tcp1 \
>   --backfstype=zfs \
>   demo-mdt0pool/mdt0 \
>   mirror sdb sdd
```

The command line formats a new MDT that will be used by the MDS for storage. The MDT will provide metadata for a file system called `demo`, and has index number 0 (zero). There are two NIDs defined as the nodes able to host the MDS service, denoted by the `--servicenode` options, and two NIDs supplied for the MGS that the MDS will register with. The command further defines a mirrored zpool called `demo-mdt0pool` consisting of two devices, and creates a ZFS dataset called `mdt0`. Normally, it is expected that the MDT will be created from a larger pool of storage, to maximize performance and required capacity; the above example is provided for the purposes of outlining the command line syntax.

The `--failnode` syntax is similar, but is used to define only a failover target for the storage service. For example:

```
[root@rh7z-mds2 system]# mkfs.lustre --mdt \
>   --fsname demo --index 0 \
>   --mgsnode 192.168.227.11@tcp1 --mgsnode 192.168.227.12@tcp1 \
>   --failnode 192.168.227.11@tcp1 \
>   --backfstype=zfs \
>   demo-mdt0pool/mdt0 mirror sdb sdd
```

Here, the failover host is identified as `192.168.227.11@tcp1`, one server in an HA pair (and which, incidentally, has the hostname `rh7z-mds1`). The `mkfs.lustre` command was executed on `rh7z-mds2` (NID: `192.168.227.12@tcp1`), and the mount command must also be run from this host when the service starts for the very first time.

Note that when creating a ZFS-based OSD using only the `mkfs.lustre` command, it is not possible to set or change the properties of the zpool, such as the `ashift` property. For this reason, it is highly recommended that the zpools be created independently of the `mkfs.lustre` command, as shown in the next section.

Formatting an MDT using `zpool` and `mkfs.lustre`

To create a ZFS-based MDT, create a zpool to contain the MDT file system dataset, then use `mkfs.lustre` to create the actual file system dataset inside the zpool:

```
zpool create [-f] -O canmount=off \
```

```
    [-o ashift=<n>] \
    -o cachefile=/etc/zfs/<zpool name>.spec | -o cachefile=none \
    <zpool name> \
    <zpool specification>

  mkfs.lustre --mdt \
    [--reformat] \
    --fsname <name> \
    --index <n> \
    --mgsnode <MGS NID> [--mgsnode <MGS NID> …] \
    [ --servicenode <NID> [--servicenode <NID> …]] \
    [ --failnode <NID> [--failnode <NID> …]] \
    --backfstype=zfs \
    [ --mkfsoptions <options> ] \
    <pool name>/<dataset>
```

For example:

```
  # Create the zpool
  # Pool will comprise 3 mirrors each with 2 devices.
  # Mirrors will be concatenated (striped).
  zpool create -O canmount=off \
    -o cachefile=none \
    demo-mdt0pool \
    mirror sdd sde mirror sdf sdg mirror sdh sdi

  # Format MDT0 for Lustre file system "demo"
  mkfs.lustre --mdt \
    --fsname demo \
    --index 0 \
    --mgsnode 192.168.227.11@tcp1 --mgsnode 192.168.227.12@tcp1 \
    --servicenode 192.168.227.12@tcp1 \
    --servicenode 192.168.227.11@tcp1 \
    --backfstype=zfs \
    demo-mdt0pool/mdt0
```

The output from the above example will look something like this:

```
  # The zpool command will not return output if there are no errors
  [root@rh7z-mds2 system]# zpool create -O canmount=off \
  >   -o cachefile=none \
  >   demo-mdt0pool \
  >   mirror sdd sde mirror sdf sdg mirror sdh sdi

  [root@rh7z-mds2 system]# mkfs.lustre --mdt \
```

```
>   --fsname demo \
>   --index 0 \
>   --mgsnode 192.168.227.11@tcp1 --mgsnode 192.168.227.12@tcp1 \
>   --servicenode 192.168.227.12@tcp1 \
>   --servicenode 192.168.227.11@tcp1 \
>   --backfstype=zfs \
>   demo-mdt0pool/mdt0

   Permanent disk data:
Target:     demo:MDT0000
Index:      0
Lustre FS:  demo
Mount type: zfs
Flags:      0x1061
              (MDT first_time update no_primnode )
Persistent mount opts:
Parameters:  mgsnode=192.168.227.11@tcp1:192.168.227.12@tcp1
failover.node=192.168.227.12@tcp1:192.168.227.11@tcp1

checking for existing Lustre data: not found
mkfs_cmd = zfs create -o canmount=off -o xattr=sa demo-mdt0pool/mdt0
Writing demo-mdt0pool/mdt0 properties
  lustre:version=1
  lustre:flags=4193
  lustre:index=0
  lustre:fsname=demo
  lustre:svname=demo:MDT0000
  lustre:mgsnode=192.168.227.11@tcp1:192.168.227.12@tcp1
  lustre:failover.node=192.168.227.12@tcp1:192.168.227.11@tcp1
```

Use the `zfs get` command or `tunefs.lustre` to verify that the file system dataset has been formatted correctly. For example:

```
[root@rh7z-mds2 ~]# zfs get all -s local
NAME                PROPERTY              VALUE                                  SOURCE
demo-mdt0pool       canmount              off                                    local
demo-mdt0pool/mdt0  canmount              off                                    local
demo-mdt0pool/mdt0  xattr                 sa                                     local
demo-mdt0pool/mdt0  lustre:svname         demo-MDT0000                           local
demo-mdt0pool/mdt0  lustre:flags          4129                                   local
demo-mdt0pool/mdt0  lustre:failover.node  192.168.227.12@tcp1:192.168.227.11@tcp1  local
demo-mdt0pool/mdt0  lustre:version        1                                      local
demo-mdt0pool/mdt0  lustre:mgsnode        192.168.227.11@tcp1:192.168.227.12@tcp1  local
demo-mdt0pool/mdt0  lustre:fsname         demo                                   local
demo-mdt0pool/mdt0  lustre:index          0                                      local
```

Starting and stopping the MDS Service

The mount command is used to start all Lustre storage services, including the MDS. The syntax is:

```
mount -t lustre [-o <options>] \
  <ldiskfs blockdev>|<zpool>/<dataset> <mount point>
```

The mount command syntax is very similar for both LDISKFS and ZFS MGT storage targets. The main difference is the format of the path to the storage. For ldiskfs, the path will resolve to a block device, such as `/dev/sda` or `/dev/mapper/mpatha`, whereas for ZFS, the path resolves to a dataset in a zpool, e.g. `demo-mdt0pool/mdt0`.

The following example starts a ZFS-based MDS:

```
# Ignore MOUNTPOINT column in output: not used by Lustre
[root@rh7z-mds2 ~]# zfs list -o name,used,avail,refer
NAME                 USED  AVAIL  REFER
demo-mdt0pool       2.87M  9.62G    19K
demo-mdt0pool/mdt0  2.79M  9.62G  2.79M

[root@rh7z-mds2 ~]# mkdir -p /lustre/demo/mdt0

[root@rh7z-mds2 ~]# mount -t lustre demo-mdt0pool/mdt0
/lustre/demo/mdt0

[root@rh7z-mds2 ~]# df -ht lustre
File system          Size  Used Avail Use% Mounted on
demo-mdt0pool/mdt0  9.7G  2.8M  9.7G   1% /lustre/demo/mdt0
```

Note that the output for `zfs list` shows the mount points for the MGS Pool and MGT dataset in the MOUNTPOINT column. Just as for all ZFS-formatted OSDs, the content in this column can be ignored.

The recommended convention for the mount point of the MDT storage is `/lustre/<fsname>/mdt<n>`, where `<fsname>` is the name of the file system and <n> is the index number of the MDT.

Remember that only the `mount -t lustre` command can start Lustre services. Mounting storage as type `ldiskfs` or `zfs` will mount a storage target on the host, but it will not trigger the startup of the requisite kernel processes.

To verify that the MDS is running, check that the device has been mounted, then get the Lustre device list with `lctl dl`, and review the running processes:

```
[root@rh7z-mds2 ~]# df -ht lustre
```

```
File system          Size  Used Avail Use% Mounted on
demo-mdt0pool/mdt0  9.7G  2.8M  9.7G   1% /lustre/demo/mdt0

[root@rh7z-mds2 ~]# lctl dl
  0 UP osd-zfs demo-MDT0000-osd demo-MDT0000-osd_UUID 7
  1 UP mgc MGC192.168.227.11@tcp1 1605562b-d702-9251-6f38-
1fd4a64e2720 5
  2 UP mds MDS MDS_uuid 3
  3 UP lod demo-MDT0000-mdtlov demo-MDT0000-mdtlov_UUID 4
  4 UP mdt demo-MDT0000 demo-MDT0000_UUID 5
  5 UP mdd demo-MDD0000 demo-MDD0000_UUID 4
  6 UP qmt demo-QMT0000 demo-QMT0000_UUID 4
  7 UP lwp demo-MDT0000-lwp-MDT0000 demo-MDT0000-lwp-MDT0000_UUID 5

[root@rh7z-mds2 ~]# ps -ef | awk '/mdt/ && !/awk/'
root     32320     2  0 Mar30 ?        00:00:00 [mdt00_000]
root     32321     2  0 Mar30 ?        00:00:00 [mdt00_001]
root     32322     2  0 Mar30 ?        00:00:00 [mdt00_002]
root     32323     2  0 Mar30 ?        00:00:00 [mdt_rdpg00_000]
root     32324     2  0 Mar30 ?        00:00:00 [mdt_rdpg00_001]
root     32325     2  0 Mar30 ?        00:00:00 [mdt_attr00_000]
root     32326     2  0 Mar30 ?        00:00:00 [mdt_attr00_001]
root     32327     2  0 Mar30 ?        00:00:00 [mdt_out00_000]
root     32328     2  0 Mar30 ?        00:00:00 [mdt_out00_001]
root     32329     2  0 Mar30 ?        00:00:00 [mdt_seqs_0000]
root     32330     2  0 Mar30 ?        00:00:00 [mdt_seqs_0001]
root     32331     2  0 Mar30 ?        00:00:00 [mdt_seqm_0000]
root     32332     2  0 Mar30 ?        00:00:00 [mdt_seqm_0001]
root     32333     2  0 Mar30 ?        00:00:00 [mdt_fld_0000]
root     32334     2  0 Mar30 ?        00:00:00 [mdt_fld_0001]
root     32340     2  0 Mar30 ?        00:00:00 [mdt_ck]
```

To stop a Lustre service, `umount` the corresponding target:

```
umount <mount point>
```

The mount point must correspond to the mount point used with the `mount -t lustre` command. For example:

```
[root@rh7z-mds2 ~]# df -ht lustre
File system          Size  Used Avail Use% Mounted on
demo-mdt0pool/mdt0  9.7G  2.8M  9.7G   1% /lustre/demo/mdt0
[root@rh7z-mds2 ~]# umount /lustre/demo/mdt0
[root@rh7z-mds2 ~]# df -ht lustre
```

```
df: no file systems processed
[root@rh7z-mds2 ~]# lctl dl
[root@rh7z-mds2 ~]#
```

Using the regular `umount` command is the correct way to stop a given Lustre service and unmount the associated storage, both ldiskfs and ZFS-based Lustre storage volumes.

Do not use the `zfs unmount` command to stop a Lustre service. Attempting to use `zfs` commands to unmount a storage target that is mounted as part of an active Lustre service will return an error.

Create the Object Storage Services (OSS)

The Object Storage Servers (OSS) in a Lustre file system provide the bulk data storage for all file content. Each OSS provides access to a set of storage volumes referred to as Object Storage Targets (OSTs) and each object storage target contains a number of binary objects representing the data for files in Lustre.

Files are composed of one or more OST objects, in addition to the metadata inode. The allocation of objects to a file is referred to as the file's layout, and is determined when the file is created. The layout for a file defines the set of OST objects that will be used to hold the file's data. Each object for a given file is held on a separate OST, and data is written to the objects in a round-robin allocation as a stripe. As an analogy, one can think of the objects of a file as virtual equivalents of disk drives in a RAID-0 array. Data is written in fixed-sized chunks in stripes across the objects. The stripe width is also configurable when the file is created, and defaults to 1MiB.

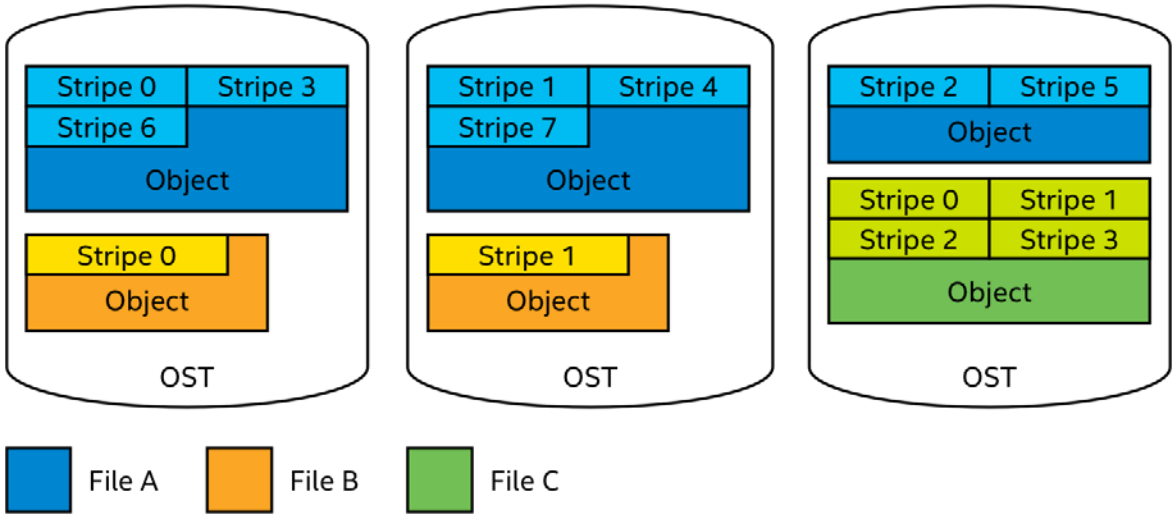

Figure 8. Files are written to one or more objects stored on OSTs in a stripe pattern

Figure 8 shows some examples of file with different storage layouts. File A is striped across 3 objects, File B comprises 2 objects and File C is a single object. The number of objects and the size of the stripe is configurable when the file is created.

The object layout specification for a file can be supplied directly by the user or application, otherwise it will be assigned by the MDT based on either the file system default layout, or by a layout policy defined for the directory in which the file has been created. The MDT is responsible for the assignment of objects to a file.

Objects for a file are created when the file is created (although for efficiency, the MDT will pre-create a pool of zero-length objects on the OSTs, ready for assignment to files as they are

created). Objects are initially empty when created and all objects for a file are created when the file itself is created; objects are not allocated dynamically as data is written. This means that if a file is created with a stripe count of 4 objects, all 4 of the objects will be allocated to the file, even if the file is initially empty, or has less than one stripe's worth of data.

Each object storage server can host several object storage targets, typically in the range 2-8 OSTs per active server, but sometimes more. There can be many object storage servers in a file system, scaling up to hundreds of servers. Each OST can be several tens of terabytes in size, and a typical OSS might serve anywhere from 100-500TB of usable capacity, depending on the storage configuration.

The OSS population therefore determines the bandwidth and overall capacity of a Lustre file system. A single file system instance can theoretically scale to 1 Exabyte of available capacity using ZFS (EXT4/ldiskfs can scale to 512PB) across hundreds of servers, and there are supercomputer installations with 50PB or more of online capacity in a single file system instance. In terms of throughput performance, there are sites that have measured sustained bandwidth in excess of 1TB/sec.

Each OST operates independently of all other OSTs in the file system and there are no dependencies between objects themselves (a file may comprise multiple objects but the objects themselves don't have a direct relationship). This, along with a flat namespace structure for objects, allows the performance of the file system to scale linearly as more OSTs are added.

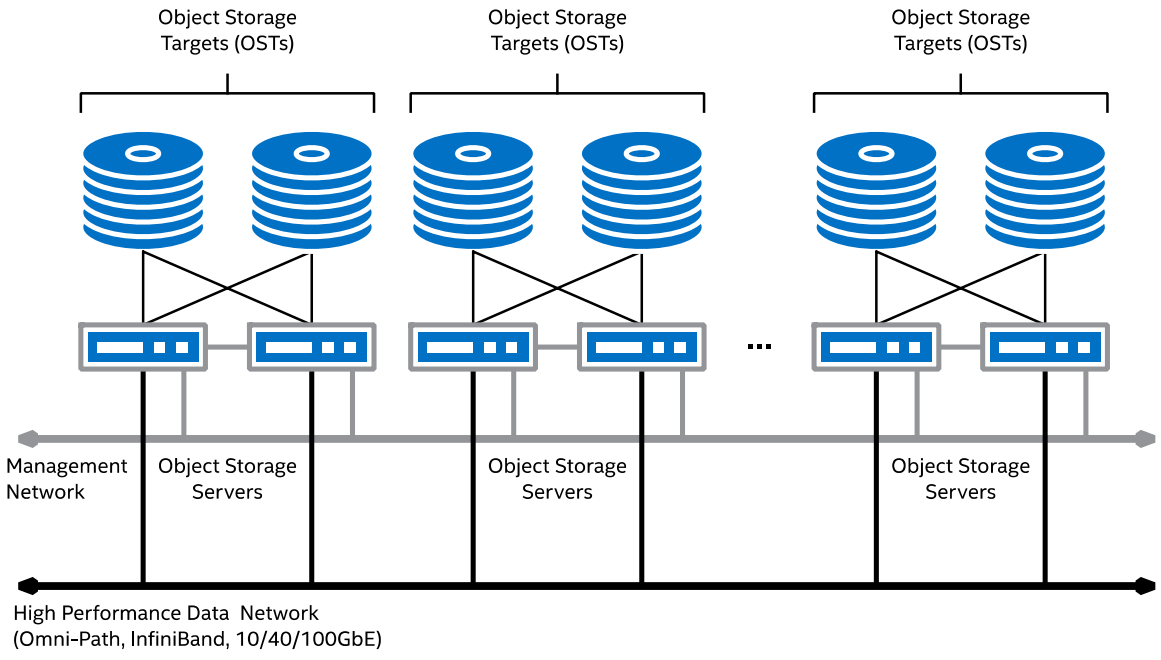

Figure 9. Performance and Capacity Scale with Linearly with Increasing Numbers of OSSs and OSTs

While there is no strict technical limitation that prevents OSTs from different file systems from being accessed from a common object storage server, it is not a common or recommended practice. As a rule, each OSS should be associated with a single Lustre file system. This simplifies system planning and maintenance, and makes the environment more predictable with regard to performance. A similar rule applies when working with high-availability configurations.

As for other Lustre server components, object storage servers can be combined into high availability configurations, and this is the normal practice for Lustre file system implementations. Unlike the metadata and management services, object storage servers are only grouped with other object storage servers when designing for HA.

A typical building block configuration will comprise two OSS hosts connected to a common pool of shared storage. This storage may be a single enclosure or several, depending on the requirements of the implementation: a site may wish to optimize the Lustre installation for throughput performance, choosing lower capacity, high performance storage devices attached to a relatively high number of servers, or it may place an emphasis on capacity over throughput with high density storage connected to relatively fewer servers. Lustre is very versatile and affords system architects flexibility in designing a solution appropriate to the requirements of the site.

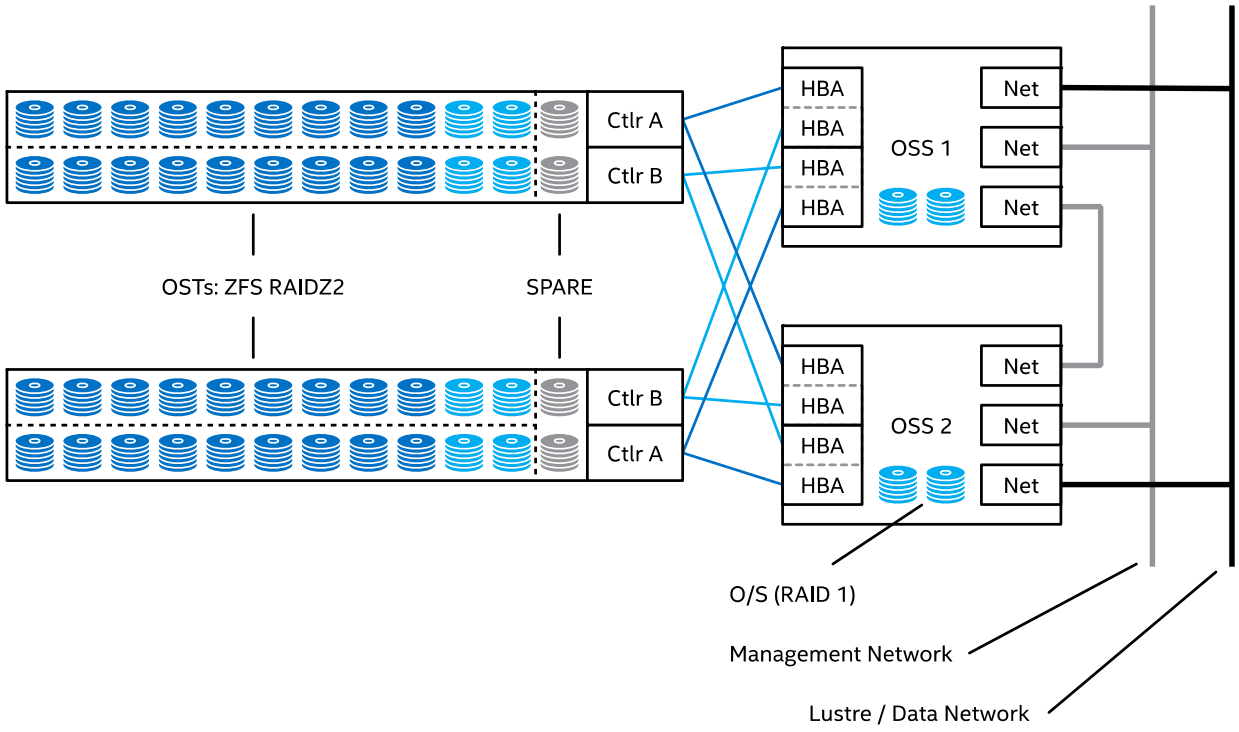

Figure 10. Object Storage Server Cluster with Four OSTs

OSS servers are throughput-oriented systems, and benefit from large memory configurations and a large number of cores for servicing Lustre kernel threads. There is less emphasis on the raw performance per core, as the workloads are not typically oriented around IOps. Throughput bandwidth is the dominant factor in designed object storage server hardware. This follows through to the storage hardware as well, which will typically comprise high-capacity, high-density enclosures. Flash storage is currently less prevalent for object storage due to the higher cost-per-terabyte of capacity.

Storage volumes are typically configured as RAID 6 volumes for ldiskfs OSDs, and RAIDZ2 for ZFS OSDs. The volumes are usually configured in anticipation of a 1MiB I/O transaction, as this is the default unit used by Lustre for all I/O. For a RAID 6 volume, this means creating a disk arrangement that can accommodate a 1MiB full stripe write without any read-modify-write overhead. Typically this means arranging the RAID 6 volumes in layouts where the number of "data disks" is a power of 2, plus the 2 parity disks. (Parity is distributed in RAID 6, so this is not an accurate representation of the data layout on the storage, but the idea is to describe the amount of usable capacity in terms of the equivalent number of disks in the RAID 6 volume.)

The most common RAID 6 layout for ldiskfs is 10 disks (8+2).

Whereas RAID 6 volume or LUN for ldiskfs will be used as an individual object storage target, ZFS OSTs will be created from a single dataset per zpool. The composition of the zpool in terms of the vdevs will require experimentation to find the optimal arrangement balancing capacity utilization and performance.

Each OST zpool must contain a minimum of one RAIDZ2 vdev, but may contain many, creating a stripe of RAIDZ2 vdevs. This arrangement is occasionally referred to as RAID 6+0 (or RAID60) when discussing non-ZFS volumes.

In theory, the same layout used for ldiskfs RAID6 (namely 10 disks, 8+2) should also apply to a ZFS RAIDZ2 vdev, but this is not always the case. While ldiskfs OSDs with RAID-6 require 8+2 for best performance, ZFS + RAID-Z2 is much more flexible and the RAID geometry should be chosen to best match the JBOD enclosure (also taking hot spare devices into account). For example, a benchmarking study conducted by Intel® Corporation found that using an 11- or 12-disk RAIDZ2 vdev configuration can, in some circumstances, yield better overall throughput results than a 10-disk allocation, particularly when working with large I/O block sizes.

Experimenting with different RAIDZ layouts, such as using 11 or 12 disks for RAIDZ2 as well as the more conventional or traditional 10 disk configuration, is therefore recommended in order to identify the optimum structure for achieving strong performance.

OST Formatted as a ZFS OSD

Formatting an OST using only the `mkfs.lustre` command

The syntax for creating a ZFS-based OST using only the `mkfs.lustre` command is as follows (Do not use this method when working with ZFS OSDs in high-availability, failover configurations):

```
mkfs.lustre --ost \
  [--reformat] \
  --fsname <name> \
  --index <n> \
  --mgsnode <MGS NID> [--mgsnode <MGS NID> …] \
  [ --servicenode <NID> [--servicenode <NID> …]] \
  [ --failnode <NID> [--failnode <NID> …]] \
  --backfstype=zfs \
  [ --mkfsoptions <options> ] \
  <pool name>/<dataset> \
  <zpool specification>
```

The basic syntax is very similar to that for the MDT: in addition to specifying the target type, the file system name and the NID[s] for the MGS are also supplied, along with the index number of the OST that is being created. The `servicenode` or `failnode` command-line options are used to identify the NIDs of the hosts that are able to run the target Lustre service in a high-availability configuration.

This example uses the `--servicenode` syntax to create an OST that can be run on two servers as an HA failover resource:

```
[root@rh7z-oss1 system]# mkfs.lustre --ost \
>   --fsname demo \
>   --index 0 \
>   --mgsnode 192.168.227.11@tcp1 --mgsnode 192.168.227.12@tcp1 \
>   --servicenode 192.168.227.21@tcp1 \
>   --servicenode 192.168.227.22@tcp1 \
>   --backfstype=zfs \
>   demo-ost0pool/ost0 \
>   raidz2 sda sdb sdc sdd sde sdf
```

The command line formats a new OST that will be used by the OSS for storage. The OST will be part of to a file system called demo, with index number 0 (zero). The back-end storage is a ZFS pool called `demo-ost0pool` comprising a RAIDZ2 vdev constructed from six physical devices, with a ZFS file system dataset called `ost0`. Two server NIDs are supplied as service

nodes for the OSS, `192.168.227.21@tcp1` and `192.168.227.22@tcp1`, and there are NIDs for the MGS primary and failover hosts.

The `failnode` syntax is similar, but is used to define only a failover target for the storage service. For example:

```
[root@rh7z-oss1 system]# mkfs.lustre --ost \
>   --fsname demo --index 0 \
>   --mgsnode 192.168.227.11@tcp1 --mgsnode 192.168.227.12@tcp1 \
>   --failnode 192.168.227.22@tcp1 \
>   --backfstype=zfs \
>   demo-ost0pool/ost0 mirror sdb sdd
```

Here, the failover host is identified as `192.168.227.22@tcp1`, an OSS server in an HA pair (and which, incidentally, has the hostname `rh7z-oss2`). The `mkfs.lustre` command was executed on `rh7z-mds1` (NID: `192.168.227.21@tcp1`), and the mount command must also be run from this host when the OSS service starts for the very first time.

Similarly to the MDT, note that when creating a ZFS-based OSD using only the `mkfs.lustre` command, it is not possible to set or change the properties of the zpool, such as the `ashift` property. For this reason, it is highly recommended that the zpools be created independently of the `mkfs.lustre` command, as shown in the next section.

Formatting an OST using `zpool` and `mkfs.lustre`

To create a ZFS-based OST, create a zpool to contain the OST file system dataset, then use `mkfs.lustre` to create the actual file system dataset inside the zpool:

```
zpool create [-f] -O canmount=off \
  [-o ashift=<n>] \
  -o cachefile=/etc/zfs/<zpool name>.spec | -o cachefile=none \
  <zpool name> \
  <zpool specification>

mkfs.lustre --ost \
  [--reformat] \
  --fsname <name> \
  --index <n> \
  --mgsnode <MGS NID> [--mgsnode <MGS NID> …] \
  [ --servicenode <NID> [--servicenode <NID> …]] \
  [ --failnode <NID> [--failnode <NID> …]] \
  --backfstype=zfs \
  [ --mkfsoptions <options> ] \
  <pool name>/<dataset>
```

For example:

```
# Create the zpool
# Pool will comprise a single RAIDZ2 vdev with 6 devices
zpool create -O canmount=off \
  -o cachefile=none \
  demo-ost0pool \
  raidz2 sda sdb sdc sdd sde sdf

# Format OST0 for Lustre file system "demo"
mkfs.lustre --ost \
  --fsname demo \
  --index 0 \
  --mgsnode 192.168.227.11@tcp1 --mgsnode 192.168.227.12@tcp1 \
  --servicenode 192.168.227.21@tcp1 \
  --servicenode 192.168.227.22@tcp1 \
  --backfstype=zfs \
  demo-ost0pool/ost0
```

The output from the above example will look something like this:

```
# The zpool command will not return output unless there are errors
[root@rh7z-oss1 system]# zpool create -O canmount=off \
>   -o cachefile=none \
>   demo-ost0pool \
>   raidz2 sda sdb sdc sdd sde sdf

[root@rh7z-oss1 lz]# mkfs.lustre --ost \
>   --fsname demo \
>   --index 0 \
>   --mgsnode 192.168.227.11@tcp1 --mgsnode 192.168.227.12@tcp1 \
>   --servicenode 192.168.227.21@tcp1 \
>   --servicenode 192.168.227.22@tcp1 \
>   --backfstype=zfs \
>   demo-ost0pool/ost0

   Permanent disk data:
Target:     demo:OST0000
Index:      0
Lustre FS:  demo
Mount type: zfs
Flags:      0x1062
              (OST first_time update no_primnode )
Persistent mount opts:
```

```
Parameters:  mgsnode=192.168.227.11@tcp1:192.168.227.12@tcp1
failover.node=192.168.227.21@tcp1:192.168.227.22@tcp1

checking for existing Lustre data: not found
mkfs_cmd = zfs create -o canmount=off -o xattr=sa demo-ost0pool/ost0
Writing demo-ost0pool/ost0 properties
  lustre:version=1
  lustre:flags=4194
  lustre:index=0
  lustre:fsname=demo
  lustre:svname=demo:OST0000
  lustre:mgsnode=192.168.227.11@tcp1:192.168.227.12@tcp1
  lustre:failover.node=192.168.227.21@tcp1:192.168.227.22@tcp1
```

Use the `zfs get` command or `tunefs.lustre` to verify that the file system dataset has been formatted correctly. For example:

```
[root@rh7z-oss1 ~]# zfs get all -s local
NAME               PROPERTY              VALUE                                  SOURCE
demo-ost0pool      canmount              off                                    local
demo-ost0pool/ost0 canmount              off                                    local
demo-ost0pool/ost0 xattr                 sa                                     local
demo-ost0pool/ost0 lustre:mgsnode        192.168.227.11@tcp1:192.168.227.12@tcp1 local
demo-ost0pool/ost0 lustre:flags          4130                                   local
demo-ost0pool/ost0 lustre:fsname         demo                                   local
demo-ost0pool/ost0 lustre:version        1                                      local
demo-ost0pool/ost0 lustre:failover.node  192.168.227.21@tcp1:192.168.227.22@tcp1 local
demo-ost0pool/ost0 lustre:index          0                                      local
demo-ost0pool/ost0 lustre:svname         demo-OST0000                           local
```

Starting and stopping the OSS Service

The `mount` command is used to start all Lustre storage services, including the OSS. The syntax is:

```
mount -t lustre [-o <options>] \
  <ldiskfs blockdev>|<zpool>/<dataset> <mount point>
```

The following example starts a ZFS-based OSS:

```
[root@rh7z-oss1 ~]# zfs list -o name,used,avail,refer
NAME                 USED  AVAIL  REFER
demo-ost0pool        173M  48.0G    19K
demo-ost0pool/ost0   173M  48.0G   173M

[root@rh7z-oss1 lz]# mkdir -p /lustre/demo/ost0
[root@rh7z-oss1 lz]# mount -t lustre demo-ost0pool/ost0
/lustre/demo/ost0
```

```
[root@rh7z-oss1 lz]# df -ht lustre
File system          Size  Used Avail Use% Mounted on
demo-ost0pool/ost0   49G  2.9M   49G   1% /lustre/demo/ost0
```

The recommended convention for the mount point of the OST storage is `/lustre/<fsname>/ost<n>`, where `<fsname>` is the name of the file system and `<n>` is the index number of the OST. As with all Lustre storage targets, only the `mount -t lustre` command can start Lustre services.

To verify that the OSS is running, check that the device has been mounted, then get the Lustre device list with `lctl dl` and review the running processes:

```
[root@rh7z-oss1 lz]# lctl dl
  0 UP osd-zfs demo-OST0000-osd demo-OST0000-osd_UUID 5
  1 UP mgc MGC192.168.227.11@tcp1 4106d169-ed51-cd92-3361-
12800a73962d 5
  2 UP ost OSS OSS_uuid 3
  3 UP obdfilter demo-OST0000 demo-OST0000_UUID 7
  4 UP lwp demo-MDT0000-lwp-OST0000 demo-MDT0000-lwp-OST0000_UUID 5

[root@rh7z-oss1 lz]# ps -ef | awk '/ost/ && !/awk/'
root      1932     1  0 Apr04 ?        00:00:00
/usr/libexec/postfix/master -w
postfix   1984  1932  0 Apr04 ?        00:00:00 pickup -l -t unix -u
postfix   1985  1932  0 Apr04 ?        00:00:00 qmgr -l -t unix -u
root     24709     2  0 00:03 ?        00:00:00 [ll_ost00_000]
root     24710     2  0 00:03 ?        00:00:00 [ll_ost00_001]
root     24711     2  0 00:03 ?        00:00:00 [ll_ost00_002]
root     24712     2  0 00:03 ?        00:00:00 [ll_ost_create00]
root     24713     2  0 00:03 ?        00:00:00 [ll_ost_create00]
root     24714     2  0 00:03 ?        00:00:00 [ll_ost_io00_000]
root     24715     2  0 00:03 ?        00:00:00 [ll_ost_io00_001]
root     24716     2  0 00:03 ?        00:00:00 [ll_ost_io00_002]
root     24717     2  0 00:03 ?        00:00:00 [ll_ost_seq00_00]
root     24718     2  0 00:03 ?        00:00:00 [ll_ost_seq00_00]
root     24719     2  0 00:03 ?        00:00:00 [ll_ost_out00_00]
root     24720     2  0 00:03 ?        00:00:00 [ll_ost_out00_00]
```

To stop a Lustre service, `umount` the corresponding target:

```
umount <mount point>
```

The mount point must correspond to the mount point used with the `mount -t lustre` command. For example:

```
[root@rh7z-oss1 lz]# df -ht lustre
File system          Size  Used Avail Use% Mounted on
demo-ost0pool/ost0   49G  2.9M   49G   1% /lustre/demo/ost0
[root@rh7z-oss1 ~]# umount /lustre/demo/ost0
[root@rh7z-oss1 ~]# df -ht lustre
df: no file systems processed
[root@rh7z-oss1 ~]# lctl dl
[root@rh7z-oss1 ~]#
```

The regular `umount` command is the correct way to stop a given Lustre service and unmount the associated storage, for both ldiskfs and ZFS-based Lustre storage volumes.

Do not use the `zfs unmount` command to stop a Lustre service. Attempting to use `zfs` commands to unmount a storage target that is mounted as part of an active Lustre service will return an error.

Lustre Clients

All end-user application I/O happens via a service called the Lustre client. The client is responsible for providing a POSIX interface to applications, creating a coherent presentation of the metadata (file system name space) and object data (file content) to applications running on the client operating system. All Lustre file system IO is transacted over a network protocol. Client specifications are entirely application-driven and vary widely across the spectrum of applications, organisations and industries. Lustre clients must be running a Linux operating system, and the software is comprised of kernel modules, with some user-space tools to assist with configuration and management.

Starting and stopping the Lustre Client

Start a Lustre client using the `mount` command, the basic syntax of which is:

```
mount -t lustre \
  [-o <options> ] \
  <MGS NID>[:<MGS NID>]:/<fsname> \
  /lustre/<fsname>
```

To stop the Lustre client, unmount the file system:

```
umount <path>
```

The `mount` and `umount` commands require super-user privileges to run.

When the `mount` command is invoked, the client first registers with the MGS to retrieve the configuration information, also referred to as the log, for the file system that it wants to mount. A single MGS can store the configuration information for more than one file system.

The following example shows the command line used to mount a file system named `demo`:

```
mkdir -p /lustre/demo
mount -t lustre \
  192.168.227.11@tcp1:192.168.227.12@tcp1:/demo \
  /lustre/demo
```

There are two MGS server NIDs supplied on the command line. The client will try to connect to the MGS in the order of the addresses supplied on the command line. If connection to the first NID fails, the client will attempt a connection using the second NID.

To verify that the file system is mounted on the client, use the `df` command:

```
[root@rh7z-c3 ~]# df -ht lustre
File system                                  Size  Used Avail
Use% Mounted on
```

```
192.168.227.11@tcp1:192.168.227.12@tcp1:/demo   49G  2.9M   49G   1%
/lustre/demo
```

The `lctl dl` command provides detail on the connections to the Lustre services:

```
[root@rh7z-c3 ~]# lctl dl
  0 UP mgc MGC192.168.227.11@tcp1 7f07b5f9-27e3-0b09-7456-
d83ae184d204 5
  1 UP lov demo-clilov-ffff8800bab6a000 c04fa65d-3f0b-9cbf-b373-
6a894da8e0be 4
  2 UP lmv demo-clilmv-ffff8800bab6a000 c04fa65d-3f0b-9cbf-b373-
6a894da8e0be 4
  3 UP mdc demo-MDT0000-mdc-ffff8800bab6a000 c04fa65d-3f0b-9cbf-
b373-6a894da8e0be 5
  4 UP osc demo-OST0000-osc-ffff8800bab6a000 c04fa65d-3f0b-9cbf-
b373-6a894da8e0be 5
```

If the MGS is unavailable, the mount command will return an error, similar to the following example:

```
[root@rh7z-c3 ~]# mount -t lustre \
>   192.168.227.11@tcp1:192.168.227.12@tcp1:/demo \
>   /lustre/demo
mount.lustre: mount 192.168.227.11@tcp1:192.168.227.12@tcp1:/demo at
/lustre/demo failed: Input/output error
Is the MGS running?
```

More detailed information on the failure will be in the syslog and kernel ring buffer:

```
[ 9996.909126] Lustre:
10199:0:(client.c:1967:ptlrpc_expire_one_request()) @@@ Request sent
has timed out for slow reply: [sent 1459822631/real 1459822631]
req@ffff8800ad298000 x1530734379532292/t0(0) o250-
>MGC192.168.227.11@tcp1@192.168.227.11@tcp1:26/25 lens 400/544 e 0
to 1 dl 1459822636 ref 1 fl Rpc:XN/0/ffffffff rc 0/-1
[10021.923403] Lustre:
10199:0:(client.c:1967:ptlrpc_expire_one_request()) @@@ Request sent
has timed out for slow reply: [sent 1459822656/real 1459822656]
req@ffff880138238000 x1530734379532308/t0(0) o250-
>MGC192.168.227.11@tcp1@192.168.227.12@tcp1:26/25 lens 400/544 e 0
to 1 dl 1459822661 ref 1 fl Rpc:XN/0/ffffffff rc 0/-1
[10027.155495] LustreError: 15c-8: MGC192.168.227.11@tcp1: The
configuration from log 'demo-client' failed (-5). This may be the
result of communication errors between this node and the MGS, a bad
configuration, or other errors. See the syslog for more information.
[10027.207044] Lustre: Unmounted demo-client
```

```
[10027.214618] LustreError:
10212:0:(obd_mount.c:1342:lustre_fill_super()) Unable to mount  (-5)
```

The most common cause of failure is an improperly configured network interface, or LNet NID. Verify that the LNet protocol is able to communicate with the MGS with `lctl ping`:

```
lctl ping <MGS NID>
```

If the ping fails, the command will return an I/O error:

```
[root@rh7z-c3 ~]# lctl ping 192.168.227.11@tcp1
failed to ping 192.168.227.11@tcp1: Input/output error
```

Check the LNet settings before continuing.

If the ping succeeds, but the mount still fails, verify that the Lustre services are running on the target host. Also check to see if there are any services running on the client that might be interfering with communication, such as a firewall or SELinux. While SELinux is supported in Lustre 2.8, and Intel EE Lustre 3.0, older releases of Lustre are not compatible. Temporarily disabling the firewall and SELinux can help narrow down the root cause of issues with Lustre communications.

If there are no OSS services online, but the MGS and the MDS for MDT0 are running, then the client mount command will hang indefinitely until an OSS service starts up.

There are options specific to Lustre that can be applied to the Lustre client mount command. The most common of these are `flock`, `localflock` and `user_xattr`:

- `flock`: enable support for cluster-wide, coherent file locks. Must be applied to the mount commands for all clients that will be accessing common data requiring lock functionality. Cluster-wide locking will have a detrimental impact on file system performance, and should only be enabled when absolutely required. For some applications, the locking is only necessary on a sub-set of nodes. For example, the CTDB cluster framework used by Samba to provide a parallel, high-availability SMB gateway, relies on locking of a shared file when coordinating cluster start-up and recovery. However, only the CTDB nodes need to mount the Lustre file system with the `flock` option. This is an example of application or domain-specific lock requirements.
- `localflock`: enable client-local flock support. This is much faster than cluster-wide flock support, but is only suitable for applications that require locks, but don't run on multiple hosts (or where the data will not be accessed in a manner that would require locking across multiple hosts).
- `user_xattr`: Enable support for user extended attributes.

Additionally, consider using the `_netdev` mount option when mounting the Lustre client, especially when adding an entry into `/etc/fstab`. This option indicates to the operating system that the file system has a dependency on the network such that it should not be mounted before the network is online and should be unmounted on shutdown prior to stopping the network stack. An example entry for `/etc/fstab`:

```
192.168.227.11@tcp1:192.168.227.12@tcp1:/demo /lustre/demo lustre defaults,_netdev 0 0
```

Refer to the `mount.lustre` man page for more information on the available options.

Starting and Stopping Lustre Services

Lustre Start-up Sequence

The normal start-up sequence for a Lustre file system is as follows:

1. Start the Management Service (MGS):

   ```
   <log into MGS host>
   # For ZFS OSDs only, must import the zpool
   zpool import <zpool name>
   mkdir -p /lustre/mgt
   mount -t lustre <path to MGT> /lustre/mgt
   ```

2. Start the Metadata Service[s] (MDS):

   ```
   <log into MDS host>
   # For ZFS OSDs only, must import the zpool
   zpool import <zpool name>
   mkdir -p /lustre/<fsname>/mdt<n>
   mount -t lustre  <path to MDTn> /lustre/<fsname>/mdt<n>
   ```

   where <n> represents the MDT index number. There must be, at a minimum, an `mdt0` storage target.

3. Start the Object Storage Services (OSS)

   ```
   <log into OSS host>
   # For ZFS OSDs only, must import the zpool
   zpool import <zpool name>
   mkdir -p /lustre/<fsname>/ost<n>
   mount -t lustre  <path to OSTn> /lustre/<fsname>/ost<n>
   ```

   where <n> represents the OST index number, starting from 0 (zero)

4. Mount the Lustre clients

   ```
   <log into client host>
   mkdir -p /lustre/<fsname>
   mount -t lustre  <MGS NID>[:<MGS NID>]:/<fsname> /lustre/<fsname>
   ```

   The MGS NIDs are the LNet network identifiers for the MGS servers and should be listed in preferred search order. The Lustre client software will attempt to connect to each NID in the order specified on the command line (or in the /etc/fstab file). The client will stop searching when it has successfully established a connection to the MGS.

Note: the clients will connect to the Management Service (MGS) first, not the Metadata Service (MDS). This is in fact common to all Lustre assets except the MGS itself, but it is more obvious with the clients when mounting a file system.

If the two services are installed on an HA cluster, make sure to list the NIDs such that the expected preferred primary node is listed first.

If there are no OSS services online, but the MGS and the MDS for MDT0 are running, then the client mount will hang indefinitely until an OSS service starts up.

Lustre Shutdown Sequence

The shutdown sequence for a Lustre file system environment is:

1. Stop all of the Lustre clients

   ```
   <Log into client host>
   umount /lustre/<fsname>
   ```

2. Stop the Metadata Service[s] (MDS)

   ```
   <Log into MDS host>
   umount /lustre/<fsname>/mdt<n>
   ```

3. Stop the Object Storage Services (OSS)

   ```
   <Log into OSS host>
   umount /lustre/<fsname>/ost<n>
   ```

4. [Optional] Stop the Management Service (MGS)

   ```
   <Log into MGS host>
   umount /lustre/mgt
   ```

Note that for ZFS OSDs, it is not necessary to export the zpool when stopping Lustre services. The services are stopped by the umount command. The zpools need only be exported when migrating the ZFS pool for import on a different host.

Why not start the MDS after the OSSs?

The metadata server is the gateway for all I/O in a Lustre file system, as it controls all of the namespace operations and is responsible for providing the layout of files across the object storage. Stopping the MDS before the OSTs prevents any new IO during shutdown, and starting the MDS after the OSTs prevents any new IO until all the services are online. In effect, files cannot be created or destroyed if the metadata service is offline.

As such it is reasonable to expect that the MDS is the last service component started in a Lustre file system and the first service that is stopped on file system shutdown. However, when considering expansion of a Lustre file system by adding more OSTs, the MDT services must be online before the new storage targets are added.

If the file system is new, then the startup sequence must be MGS → MDS → OSS → Client, and if a new OST is to be added to the file system, the MDS *must* be online before the new OST is started.

A startup sequence of MGS → OSS → MDS → Client is acceptable *only if the file system is established and the server configuration is static*. New storage targets can only be added to Lustre when the MGS and MDT0 are online.

High Availability and Failover

Controlling Service Failover Between Hosts

Service continuity, or high-availability of a service, is implemented in Lustre by means of a mechanism called "failover". A failover service is one that can be run in exactly one location at a time, but has a choice of hosts on which to run. The hosts are usually identically similar in configuration, and have some common infrastructure characteristics that permit the service to start and stop in a predictable manner on each host. The service is therefore able to run with the same configuration and data, regardless of the selected host. If the service is running on a host that fails, it can be restarted on one of the surviving hosts in the common infrastructure pool.

Lustre services are tightly coupled to the data storage. An MGS has a corresponding MGT, each MDS has one or more corresponding MDTs, and each OSS has one or more corresponding OSTs. Failover resources are defined in terms of the storage targets (MGT, MDTs, OSTs). It is essentially the storage that migrates, or "fails over", when a host develops a fault. This is because Lustre services are governed by the `mount` and `umount` commands. There are no user-space daemons like `httpd` or `nfsd` to start and stop; we simply mount the Lustre storage target to start the service, and unmount the target to stop the service.

This also means that the storage targets must be reachable by more than one server. Typically, Lustre file systems are assembled from multiple pairs of servers, each pair being connected to shared external storage. Each host in a pair mounts a subset of the connected storage, usually presented in the form of one or more LUNs (an aggregated set of devices arranged in a RAID pattern or stripe) or, in the case of JBOD storage enclosures, discrete devices.

For example, in a typical metadata server cluster pair, the storage will comprise one MGT volume and one MDT volume. One server will act as the primary host for the MGT and corresponding MGS, while the other server is the primary or preferred host for the MDT/MDS. If there is a server failure, the affected services are restarted on the surviving host.

For failover to work with Lustre, the storage targets must be configured with the NIDs of the hosts that are expected to be able to mount and provide storage services for that specific storage target. These are specified using either the `--failnode` or `--servicenode` command line options to `mkfs.lustre`.

For high-availability clusters with ZFS storage shared between nodes as a failover resource, it is also required that each zpool is created and imported with the `cachefile` property set equal to the special value of `none`:

```
zpool create [-f] -O canmount=off -o cachefile=none \
  [-o ashift=<n>] \
```

```
  <zpool name> <zpool specification>

zpool import [-f] -o cachefile=none <zpool name>
```

Controlled Migration or Failover of a Lustre Service Between Hosts

The migration, or "failover", of a Lustre service between hosts in a cooperative high-availability cluster is straightforward. A Lustre service runs wherever the corresponding storage is mounted (e.g., the MGS runs where the MGT has been mounted, similarly for MDS/MDT or OSS/OST). A service is started when the storage targets are mounted, and stopped when the storage targets are unmounted. So, we migrate a Lustre service by unmounting the storage from one host and remounting it on a different host in the cluster framework.

The concept applies to both LDISKFS and ZFS object storage devices (OSDs). For ZFS storage, there is some additional complexity due to the fact that ZFS is a volume management solution as well as a file system, and because some care is required in the process to prevent storage from being mounted on multiple hosts simultaneously.

The procedure for the controlled migration of a ZFS storage target, where both hosts are online and active, is as follows:

1. Unmount the storage from the current (primary) host:

   ```
   umount <path>
   ```
   or
   ```
   zfs unmount <path>
   ```

2. Use the `zpool export` command to remove the zpool from the configuration on the current host:

   ```
   zpool export <zpool name>
   ```

3. Log into the failover host.

4. Run the `zpool import` command, being sure to set `cachefile=none`:

   ```
   zpool import [-f] -o cachefile=none <zpool name>
   ```

   Do not use the `-f` flag unless absolutely necessary. The zpool should import cleanly if it was exported from the primary host. If the import fails, and the output from `zpool import` makes reference that the zpool may be in use on another host, check the host that is referred to. Ensure that the zpool has been properly exported from the primary host by running `zpool list` on the primary and verifying that the pool is no longer present in the listed output.

   If the pool is positively confirmed as being exported or at least not active on any other host, then run the `zfs import` command again, including the `-f` flag.

5. Check that the zpool is imported cleanly and that there are no active issues on the pool:

   ```
   zpool status
   zfs list
   ```

6. Mount the storage on the failover host:

   ```
   mkdir -p <mount point>
   mount -t lustre <zpool name>/<dataset name> <mountpoint>
   ```

7. Verify that the services are running:

   ```
   df -ht lustre
   lctl dl
   ```

The failover host is now effectively the new primary for the migrated storage. To migrate the service back to the original host, run the same procedure, with the roles of the hosts reversed (original failover host is the new primary, and the original primary host is the new failover).

Forced Migration of a Lustre Service Between Hosts

If the primary host has failed, and it is not possible to log into the host or otherwise unmount the ZFS storage and export the zpool, then a forced migration must be undertaken in order to restore service using the failover node.

The process for a forced migration is very similar to a controlled migration, but has no interaction with the original primary host, because the primary host is offline:

1. Remove power from the failed node or otherwise ensure that it is unable to render access to the shared storage containing the ZFS file systems.
2. Log into the failover host.

   Be absolutely certain that no other host has imported the dataset before continuing. The `zdb` command can be useful in verifying the on disk configuration:

   ```
   zdb -e <zpool name>
   ```

   For a two-node HA failover group, if the `hostname` and `hostid` fields match the identity of the host that has the fault, and the faulted host has been isolated from the storage (e.g. powered off), then it is safe to proceed with the import. If there were more than two hosts connected to the shared storage, make sure that no other host has imported the zpool before continuing.
3. Run the `zpool import` command, being sure to set `cachefile=none`:

   ```
   zpool import -f -o cachefile=none <zpool name>
   ```

In this scenario, with the primary host offline, the `-f` flag will almost certainly have to be used to successfully import the zpool.

4. Check that the zpool has been imported cleanly and that there are no active issues on the pool:

   ```
   zpool status
   zfs list
   zfs get all <zpool name>[/<dataset name>]
   zdb -C <zpool name>[/<dataset name>]
   ```

   The `zdb` output will show the updated `hostname` and `hostid` fields with values corresponding to the failover host.

5. Mount the storage on the failover host:

   ```
   mkdir -p <mount point>
   mount -t lustre <zpool name>/<dataset name> <mountpoint>
   ```

6. Verify that the services are running:

   ```
   df -ht lustre
   lctl dl
   ```

To restore the service back to its original host, run through the controlled migration process for active hosts.

Note: remember that setting the SPL hostid and the ZFS pool cachefile properties correctly are both critical to protecting the ZFS storage pools from data corruption in high-availability clusters**.**

Caution: *Do not rely on the system defaults when working with shared ZFS storage in high-availability clusters***.**

High Availability Automation – Pacemaker and Corosync

High availability, usually abbreviated to "HA", is a term used to describe systems and software frameworks that are designed to preserve application service availability even in the event of a failure of a component of the system. The failed component might be software or hardware; the HA framework will attempt to respond to the failure such that the applications running within the framework continue to operate correctly.

While the number of discrete failure scenarios that might be catalogued is potentially very large, they generally fall into one of a very small number of categories:

1. Failure of the application providing the service itself
2. Failure of a software dependency upon which the application relies

3. Failure of a hardware dependency upon which the application relies
4. Failure of an external service or infrastructure component upon which the application or supporting framework relies

HA systems protect application availability by grouping sets of servers and software into cooperative units or clusters. HA clusters are typically groups of two or more servers, each running their own operating system, that communicate with one another over a network connection. HA clusters will often have multi-ported, shared external storage, with each server in the cluster connected over redundant storage paths to the storage hardware.

A cluster software framework provides communication between the cluster participants (nodes). The framework will communicate the health of system hardware and application services between the nodes in the cluster and provide means to manage services and nodes, as well as react to changes in the cluster environment (e.g., server failure).

HA systems are characterized as typically having redundancy in the hardware configuration: two or more servers, each with two or more storage IO paths and often two or more network interfaces configured using bonding or link aggregation.

Measurements of availability are normally applied to the availability of the applications running on the HA cluster, rather than the hosting infrastructure. For example, loss of a physical server due to a component failure would trigger a failover or migration of the services that the server was providing to another node in the cluster. In this scenario, the outage duration would be the measure of time taken to migrate the applications to another node and restore the applications to running state. The service may be considered degraded until the failed component is repaired and restored, but the HA framework has avoided an ongoing outage.

On systems running an operating system based on Linux, the most commonly used HA cluster framework comprises two software applications used in combination: Pacemaker – used to provide resource management – and Corosync – used to provide cluster communications and low-level management, such as membership and quorum. Pacemaker can trace its genesis back to the original Linux HA project, called Heartbeat, while Corosync is derived from the OpenAIS project.

Pacemaker and Corosync are widely supported across the major Linux distributions, including Red Hat Enterprise Linux and SuSE Linux Enterprise Server. Red Hat Enterprise Linux version 6 used a very complex HA solution incorporating several other tools, although this has been simplified since the release of RHEL 6.4. Even so, RHEL 6 installations are complex and require additional packages. Fortunately, with the release of RHEL 7, the high-availability framework from Red Hat has been rationalized around Pacemaker and Corosync version 2, simplifying the software environment. Red Hat also provides a command line tool called PCS (Pacemaker and Corosync Shell) that is available for both RHEL version 6 and version 7. PCS unifies

system management for the high availability software and abstracts the underlying software implementation by providing a common command interface.

Note that Lustre does not absolutely need to be incorporated into an HA software framework such as Pacemaker, but doing so enables the operating platform to automatically make decisions about failover/migration of services without operator intervention. HA frameworks also help with general maintenance and management of application resources.

Red Hat Enterprise Linux HA Framework Configuration for Two- Node Cluster

Red Hat Enterprise Linux version 6 has a complex history with regard to the development and provision of HA software. Prior to version 6.4, Red Hat's high availability software was difficult to install and maintain and comprised an almost bewildering array of components and configuration tools. With the release of RHEL 6.4 and in all subsequent RHEL 6 updates, this has been consolidated around three principal packages: Pacemaker, Corosync version 1, and CMAN. The software stack was further simplified in RHEL 7 to just Pacemaker and Corosync version 2.

Red Hat EL 6 HA clusters use Pacemaker to provide cluster resource management (CRM), while CMAN is used to provide cluster membership and quorum services. Corosync provides communications but no other services. CMAN is unique to Red Hat Enterprise Linux and is part of an older framework. In RHEL 7, CMAN is no longer required and its functionality is entirely accommodated by Corosync version 2, but for any HA clusters running RHEL 6, Red Hat stipulates the use of CMAN in Pacemaker clusters.

The PCS application (Pacemaker and Corosync Shell) was also introduced in RHEL 6.4 and is available in current releases of both RHEL 6 and 7. PCS simplifies the installation and configuration of HA clusters in Red Hat.

Hardware and Server Infrastructure Prerequisites

This guide will demonstrate how to configure a Lustre high-availability building block using two servers and a dedicated external storage array that is connected to both servers. This design is in keeping with the standard blueprint for Lustre server components and is a suitable basis for deployment of a production-ready, high-availability Lustre parallel file system cluster.

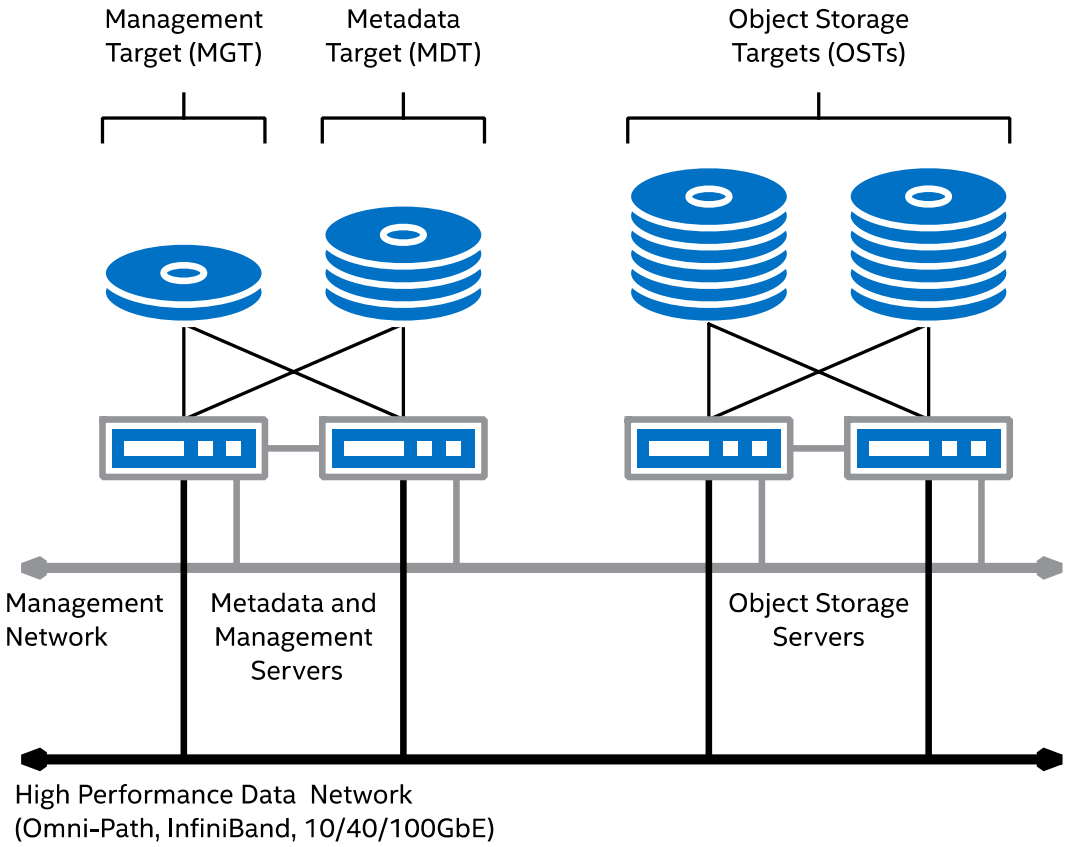

Figure 11. Lustre Server High-Availability Building Blocks

Each server depicted above requires three network interfaces:

1. A dedicated *cluster communication network* between paired servers, used as a Corosync communications ring. This can be a cross-over / point-to-point connection, or can be made via a switch.
2. A *management network* or public interface connection. This will be used by the HA cluster as an additional communications ring for Corosync.
3. Public interface, used for connection to the *high performance data network* – this the network from which Lustre services will normally be accessed by client computer systems

An alternative architecture, not specifically covered in this guide, has a single Corosync communications ring made from two network interfaces that are configured into a bond on a private network. The bond is created per the operating system documented process, and then added to the Corosync configuration just as for a discrete network interface.

Software Prerequisites

In addition to the prerequisites previously described for Lustre, the operating system requires installation of the HA software suite. It may also be necessary to enable the optional repository. For RHEL systems, the `subscription-manager` command can be used to enable the software entitlements for the HA software packages. For example:

```
subscription-manager repos \
  --enable rhel-ha-for-rhel-7-server-rpms \
  --enable rhel-7-server-optional-rpms
```

or:

```
subscription-manager repos \
  --enable rhel-ha-for-rhel-6-server-rpms \
  --enable rhel-6-server-optional-rpms
```

Refer to the documentation for the operating system distribution for more complete information on enabling subscription entitlements.

Install the HA software

1. Login as the super-user (`root`) on each of the servers in the proposed cluster and install the HA framework software:

   ```
   yum -y install pcs pacemaker corosync fence-agents [cman]
   ```

   Note: The `cman` package is only required for RHEL 6 servers.

2. On each server, add a user account to be used for cluster management and set a password. The convention is to create a user account with the name `hacluster`. The `hacluster` user should have been installed as part of the package installation (the account is created during installation of the `pacemaker-libs` package). PCS will make use of this account to facilitate cluster management: the `hacluster` account is used to authenticate the command line application, `pcs`, with the `pcsd` configuration daemon running on each cluster node. (`pcsd` is used by the `pcs` application to manage distribution of commands and synchronize the cluster configuration between the nodes.)

   The following is taken from the pacemaker-libs package postinstall script and shows the basic procedure for adding the hacluster account if it does not already exist:

   ```
   getent group haclient >/dev/null || groupadd -r haclient -g 189
   getent passwd hacluster >/dev/null || useradd -r -g haclient -u 189 -s
   /sbin/nologin -c "cluster user" hacluster
   ```

3. Set a password for the `hacluster` account. This must be set, and there is no default. Make the password *the same on each cluster node*:

   ```
   passwd hacluster
   ```

4. Modify or disable the IPTables firewall on each server in the cluster. According to Red Hat, the following ports need to be enabled:

   - TCP: ports 2224, 3121, 21064
   - UDP: ports 5405

5. In RHEL 7, the firewall software can be configured to permit cluster traffic as follows:

   ```
   firewall-cmd --permanent --add-service=high-availability
   firewall-cmd --add-service=high-availability
   ```

6. Verify the firewall configuration as follows:

   ```
   firewall-cmd --list-service
   ```

7. Alternatively, disable the firewall completely. For RHEL 7:

   ```
   systemctl stop firewalld
   systemctl disable firewalld
   ```

8. And for RHEL 6:

   ```
   chkconfig iptables off
   chkconfig ip6tables off
   service iptables stop
   service ip6tables stop
   ```

9. Start the Pacemaker configuration daemon, `pcsd`, on all servers:

   - RHEL 7: `systemctl start pcsd.service`
   - RHEL 6: `service pcsd start`

10. Verify that the service is running:

    - RHEL 7: `systemctl status pcsd.service`
    - RHEL 6: `service pcsd status`

    The following example is taken from a server running RHEL 7:

```
[root@rh7z-mds1 ~]# systemctl start pcsd.service
[root@rh7z-mds1 ~]# systemctl status pcsd.service
● pcsd.service - PCS GUI and remote configuration interface
```

```
   Loaded: loaded (/usr/lib/systemd/system/pcsd.service; enabled;
vendor preset: disabled)
   Active: active (running) since Wed 2016-04-13 01:30:52 EDT; 1min
11s ago
 Main PID: 29343 (pcsd)
   CGroup: /system.slice/pcsd.service
           ├─29343 /bin/sh /usr/lib/pcsd/pcsd start
           ├─29347 /bin/bash -c ulimit -S -c 0 >/dev/null 2>&1 ;
/usr/bin/ruby -I/usr/lib/pcsd /u...
           └─29348 /usr/bin/ruby -I/usr/lib/pcsd
/usr/lib/pcsd/ssl.rb

Apr 13 01:30:50 rh7z-mds1 systemd[1]: Starting PCS GUI and remote
configuration interface...
Apr 13 01:30:52 rh7z-mds1 systemd[1]: Started PCS GUI and remote
configuration interface.
```

11. Set up PCS authentication by executing the following command on just one of the cluster nodes:

    ```
    pcs cluster auth <node 1> <node 2> [...] -u hacluster
    ```

    For example:

    ```
    [root@rh7z-mds1 ~]# pcs cluster auth \
    >   rh7z-mds1.lfs.intl rh7z-mds2.lfs.intl \
    >   -u hacluster
    Password:
    rh7z-mds2.lfs.intl: Authorized
    rh7z-mds1.lfs.intl: Authorized
    ```

    When working with hostnames in Pacemaker and Corosync, it is recommended that the fully qualified domain name be used to reference cluster nodes.

Configure the Basic HA Framework

The `pcs` command syntax is comprehensive, but not all of the functionality may be available for RHEL 6 clusters. For example, the syntax for configuring the redundant ring protocol (RRP) for Corosync communications has only recently been added to RHEL 6. Unless otherwise stated, the commands in this section need only be executed on one node in the cluster.

The command line syntax is as follows:

```
pcs cluster setup [ --start ] --name <cluster name> \
  <node 1 specification> <node 2 specification> \
  [ --transport {udpu|udp} ] \
  [ --rrpmode {active|passive} ] \
```

```
[ --addr0 <address> ] [ --addr1 <address> ] \
[ --mcast0 <address> ] [ --mcastport0 <port> ] \
[ --mcast1 <address> ] [ --mcastport1 <port> ] \
[ --token <timeout> ] [ --join <timeout> ] \
[ ... ]
```

The node specification is a comma-separated list of hostnames or IP addresses for the host interfaces that will be used for Corosync's communications. The cluster name is an arbitrary string and will default to `pcmk` if the option is omitted.

The minimum requirement is for a single interface in the cluster configuration to be used by the cluster framework, but further interfaces can be added in order to increase the robustness of the HA cluster's inter-node messaging. Communications are organized into rings, with each ring representing a separate network. Corosync enables multiple rings using a feature called the Redundant Ring Protocol (RRP).

There are two transport types supported by the PCS command: `udpu` (UDP unicast) and `udp` (used for multicast). It is recommended that the `udp` transport be used, as it is more efficient. `udpu`, which is the default if no transport is specified, should only be selected for circumstances where multicast cannot be used.

The rings for `udpu` are determined by the node specification, which is a comma-separated list of hostnames or IP addresses associated with the ring interfaces. For example:

```
pcs cluster setup --name demo node1-A,node1-B node2-A,node2-B
```

When the `udp` transport is chosen, communications rings are defined by listing the networks upon which the Corosync multicast traffic will be carried, along with an optional list of the multicast addresses and ports that will be used. The rings are specified using the flags --addr0 and --addr1, for example:

```
pcs cluster setup --name demo node1-A node2-A --transport udp \
--addr0 10.70.0.0 --addr1 192.168.227.0
```

Use network addresses rather than host IP addresses for defining the `udp` interfaces, as this will allow a common Corosync configuration to be used across all cluster nodes. If host IP addresses are used, additional manual configuration of Corosync will be required on at least one of the cluster nodes. PCS really only works seamlessly when network addresses are used.

Corosync cannot parse network addresses supplied in the CIDR (Classless Inter-Domain Routing) notation, e.g., 10.70/16. Instead, use the full dot notation for specifying networks, e.g. 10.70.0.0 or 192.168.227.0.

The multicast addresses default to `239.255.1.1` for `ring0` and `255.239.2.1` for `ring1`. The default multicast port is `5405` for both multicast rings. Corosync actually uses two multicast ports for communication in each ring. Ports are assigned in receive / send pairs,

but only the receive port number is specified when configuring the cluster. The send port is one less than the receive port number (i.e. send port = mcastport - 1). Make sure that there is a gap of at least 1 between assigned ports for a given multicast address in a subnet. Also, if there are several HA clusters with Corosync rings on the same subnet, each cluster will require a unique multicast port pair (different clusters can use the same multicast address, but not the same multicast ports).

For example, if there are six OSSs configured into three HA pairs, and an MDS pair, then each pair of servers will require a unique multicast port for each ring, and that there must be a gap of at least one between the port numbers. So, a range of 49152, 49154, 49156, 49158 might be suitable, provided there are no other services using port 49151 (since that will be used for a send multicast port by the cluster assigned to use 49152 as the receive port). A range of 49152, 49153, 49154, 49155 is not valid because there are no gaps between the numbers to accommodate the send port.

The redundant ring protocol (RRP) mode is specified by the `--rrpmode` flag. Valid options are: `none`, `active` and `passive`. If only one interface is defined, then `none` is automatically selected. If multiple rings are defined, one of `active` or `passive` must be used.

When set to `active`, Corosync will send all messages across all interfaces simultaneously. Throughput is not as fast but overall latency is improved, especially when communicating over faulty networks.

The `passive` setting tells Corosync to use one interface, with the remaining interfaces available as standbys. If the interface fails, one of the remaining interfaces will be used instead. This is also the default mode when creating an RRP configuration with `pcs`.

The `active` mode in theory provides better reliability across multiple interfaces, while `passive` mode may be preferred when the messaging rate is more important. However, the manual page for PCS makes the choice clear and straightforward: only `passive` mode is supported by PCS and it is the only mode that receives testing.

The `--token` flag specifies the timeout in milliseconds after which a token is declared lost. If no token is received within this interval, it is lost. The default is 1000 (1000ms or 1 second). The value represents the overall length of time before a token is declared lost. Any retransmits occur within this window.

On a Lustre server cluster, the default token timeout is generally too short to accommodate variation in response when servers are under heavy load. An otherwise healthy server that is busy can take longer to pass the token to the next server in the ring than when it is idle; if the timeout is too short, the cluster might declare the token lost. Too many lost tokens from one node and the cluster will consider the node dead.

It is recommended that the value of the token parameter be increased significantly from the default. 17000ms is a reasonable, conservative value, but users will want to experiment to find the optimal setting. If the cluster seems to failover too frequently under load, but without any

other symptoms, the value should be increased as a first step to see if it alleviates the problem.

The following example uses the simplest invocation to create a cluster framework configuration comprising two nodes. This example does not specify a transport, so the default of `udp` is chosen for cluster communications:

```
pcs cluster setup --name demo-MDS \
  rh7z-mds1.lfs.intl rh7z-mds2.lfs.intl
```

The next example again uses `udpu` but incorporates a second, redundant, ring:

```
pcs cluster setup --name demo-MDS-1-2 \
  rh7z-mds1.lfs.intl,192.168.227.11 \
  rh7z-mds2.lfs.intl,192.168.227.12
```

The hostname specification is comma-separated and the node interfaces are specified in ring priority order. The first interface in the list will join `ring0`, the second interface will join `ring1`. In the above example, the `ring0` interfaces correspond to the hostname `rh7z-mds1` for the first node, and `rh7z-mds2` for the second node. The `ring1` interfaces are `192.168.227.11` and `192.168.227.12` for node 1 and node 2 respectively. One could also add the IP addresses for ring1 into the hosts table or DNS if it is preferred that the interfaces be referred to by name rather than by address.

This final example demonstrates the syntax for creating a two-node cluster with two Corosync communications rings using udp and multicast:

```
pcs cluster setup --name demo-MDS-1-2 \
  rh7z-mds1.lfs.intl rh7z-mds2.lfs.intl \
  --transport udp --rrpmode passive \
  --token 17000 \
  --addr0 10.70.0.0 --addr1 192.168.227.0 \
  --mcast0 239.255.1.1 --mcastport0 49152 \
  --mcast1 239.255.2.1 --mcastport1 49152
```

The above example will create different results when run on RHEL 6 versus RHEL 7. This is because RHEL 6 uses an additional package called CMAN, which assumes some of the responsibilities that on RHEL 7 are managed entirely by Corosync. Because of this difference, RHEL 6 clusters may behave differently to RHEL 7 clusters, even though the commands used to configure each might be identical.

If there are any unexpected or unexplained side-effects when running with RHEL 6 clusters, it is recommended that the configuration be pared down. For example, change the RRP configuration from `udp` multicast to the simpler `udpu` unicast configuration and use the comma-separated syntax to define the node addresses, rather than using the `--addr[0,1]` flags.

Changing the default security key

Changing the default key used by Corosync and CMAN for communications is optional, but will improve the overall security of the cluster installation. The different operating system distributions and releases have different procedures for managing the cluster framework authentication key, so the following information is provided for information only. Refer to the OS vendor's documentation for up to date instructions.

The default key can be changed by running the command `corosync-keygen`. The key will be written to the file `/etc/corosync/authkey`. Run the command on a single host in the cluster, then copy the resulting key to each node. The file must be owned by the root user and given read-only permissions. Example output follows:

```
[root@rh7z-mds1 ~]# corosync-keygen
Corosync Cluster Engine Authentication key generator.
Gathering 1024 bits for key from /dev/random.
Press keys on your keyboard to generate entropy.
Writing corosync key to /etc/corosync/authkey.
[root@rh7z-mds1 ~]# ll /etc/corosync/authkey
-r-------- 1 root root 128 Apr 13 23:48 /etc/corosync/authkey
```

Note that if the key is not the same for every node in the cluster, then they will not be able to communicate with each other to form a cluster. For hosts running Corosync version 2, creating the key and copying to all the nodes should be sufficient.

For hosts running RHEL 6 with the CMAN software, the cluster framework also needs to be made aware of the new key:

```
ccs -f /etc/cluster/cluster.conf \
  --setcman keyfile="/etc/corosync/authkey"
```

Starting and Stopping the cluster framework

To start the cluster framework, issue the following command from one of the cluster nodes:

```
pcs cluster start [ <node> [<node> …] | --all ]
```

To start the cluster framework on the current node only, run the `pcs cluster start` command without any additional options. To start the cluster on all nodes, supply the --all flag, and to limit the startup to a specific set of nodes, list them individually on the command line.

To shut down part or all of the cluster framework, issue the pcs stop command:

```
pcs cluster stop [ <node> [<node> …] | --all ]
```

The parameters for the `pcs stop` command are the same as for `pcs start`.

Do not configure the cluster software to run automatically on system boot. If an error occurs during the operation of the cluster and a node is isolated and powered off as a consequence, it is imperative that the node be repaired, reviewed and restored to a healthy state before committing it back to the cluster framework. Until the root cause of the fault has been isolated and corrected, adding a node back into the framework may be dangerous and could put services and data at risk.

For this reason, ensure that the pacemaker and corosync services are disabled in the sysvinit or systemd boot sequences:

RHEL 7:

```
systemctl disable corosync.service
systemctl disable pacemaker.service
```

RHEL 6:

```
chkconfig cman off
chkconfig corosync off
chkconfig pacemaker off
```

However, it is safe to keep the PCS helper daemon, `pcsd`, enabled.

Verify cluster configuration and status

To view overall cluster status:

```
pcs status [ <options> | --help]
```

For example:

```
[root@rh7z-mds1 ~]# pcs status
Cluster name: demo-MDS-1-2
WARNING: no stonith devices and stonith-enabled is not false
Last updated: Thu Apr 14 00:58:29 2016        Last change: Wed Apr
13 21:16:13 2016 by hacluster via crmd on rh7z-mds1.lfs.intl
Stack: corosync
Current DC: rh7z-mds1.lfs.intl (version 1.1.13-10.el7_2.2-44eb2dd) -
partition with quorum
2 nodes and 0 resources configured

Online: [ rh7z-mds1.lfs.intl rh7z-mds2.lfs.intl ]

Full list of resources:


PCSD Status:
```

```
    rh7z-mds1.lfs.intl: Online
    rh7z-mds2.lfs.intl: Online

  Daemon Status:
    corosync: active/disabled
    pacemaker: active/disabled
    pcsd: active/enabled
```

To review the cluster configuration:

```
pcs cluster cib
```

The output will be in the CIB XML format.

The Corosync configuration can also be reviewed:

- RHEL 7 / Corosync v2: `corosync-cmapctl`
- RHEL 6 / Corosync v1: `corosync-objctl`

This can be very useful when verifying specific changes to the cluster communications configuration, such as the RRP setup. For example:

```
[root@rh7z-mds1 ~]# corosync-cmapctl | grep interface
totem.interface.0.bindnetaddr (str) = 10.70.0.0
totem.interface.0.mcastaddr (str) = 239.255.1.1
totem.interface.0.mcastport (u16) = 49152
totem.interface.1.bindnetaddr (str) = 192.168.227.0
totem.interface.1.mcastaddr (str) = 239.255.2.1
totem.interface.1.mcastport (u16) = 49152
```

To check the status of the Corosync rings:

```
[root@rh7z-mds1 ~]# corosync-cfgtool -s
Printing ring status.
Local node ID 1
RING ID 0
   id    = 10.70.227.11
   status     = ring 0 active with no faults
RING ID 1
   id    = 192.168.227.11
   status     = ring 1 active with no faults
```

To get the cluster status from CMAN on RHEL 6 clusters:

```
[root@rh6-mds1 ~]# cman_tool status
Version: 6.2.0
```

```
Config Version: 14
Cluster Name: demo-MDS-1-2
Cluster Id: 28594
Cluster Member: Yes
Cluster Generation: 24
Membership state: Cluster-Member
Nodes: 2
Expected votes: 1
Total votes: 2
Node votes: 1
Quorum: 1
Active subsystems: 9
Flags: 2node
Ports Bound: 0
Node name: rh6-mds1.lfs.intl
Node ID: 1
Multicast addresses: 239.255.1.1 239.255.2.1
Node addresses: 10.70.206.11 192.168.206.11
```

If the cluster appears to start, but there are errors reported by `pcs cluster status` and in the syslog related to Corosync totem, then there may be a conflict in the multicast address configuration with another cluster or service on the same subnet. A typical error in the syslog would look similar to the following output:

```
Apr 13 22:11:15 rh67-pe corosync[26370]:   [TOTEM ] Received message
has invalid digest... ignoring.
Apr 13 22:11:15 rh67-pe corosync[26370]:   [TOTEM ] Invalid packet
data
```

These errors indicate that the node has intercepted traffic intended for a node on a different cluster.

Also be careful in the definition of the network and multicast addresses. PCS will often create the configuration without complaint, and the cluster framework may even load without reporting any errors to the command shell. However, a misconfiguration may lead to a failure in the RRP that it not immediately obvious. Look for unexpected information in the Corosync database and the cluster CIB. For example, if one of the cluster node addresses shows up as localhost or 127.0.0.1, this indicates a problem with the addresses supplied to `pcs` with the `--addr0` or `--addr1` flags.

The pcs status command may error out if there is only one node in the cluster:

```
[root@rh67-pe ~]# pcs cluster status
Cluster Status:
 Last updated: Wed Apr 13 22:09:21 2016
 Last change: Wed Apr 13 22:09:07 2016
```

```
 Current DC: NONE
 1 Nodes configured
 0 Resources configured

PCSD Status:
Traceback (most recent call last):
  File "/usr/sbin/pcs", line 155, in <module>
    main(sys.argv[1:])
  File "/usr/sbin/pcs", line 137, in main
    cluster.cluster_cmd(argv)
  File "/usr/lib/python2.6/site-packages/pcs/cluster.py", line 47,
in cluster_cmd
    cluster_gui_status([],True)
  File "/usr/lib/python2.6/site-packages/pcs/cluster.py", line 224,
in cluster_gui_status
    bad_nodes = check_nodes(nodes, "  ")
  File "/usr/lib/python2.6/site-packages/pcs/cluster.py", line 278,
in check_nodes
    pm_nodes = utils.getPacemakerNodesID(True)
  File "/usr/lib/python2.6/site-packages/pcs/utils.py", line 1970,
in getPacemakerNodesID
    pm_nodes[node_info[0]] = node_info[1]
IndexError: list index out of range
```

If a second node is added, or the cluster is destroyed and recreated it with two nodes, this error does not appear.

The syslog may report an error similar to the following when the cluster is first started:

```
Apr 14 01:37:05 rh6-mds1 pengine[11209]:   notice:
process_pe_message: Configuration ERRORs found during PE processing.
Please run "crm_verify -L" to identify issues.
```

By following the suggested instructions, the root cause is identified:

```
[root@rh6-mds1 ~]# crm_verify -L
Errors found during check: config not valid
  -V may provide more details
[root@rh6-mds1 ~]# crm_verify -LV
   error: unpack_resources:   Resource start-up disabled since no
STONITH resources have been defined
   error: unpack_resources:   Either configure some or disable
STONITH with the stonith-enabled option
   error: unpack_resources:   NOTE: Clusters with shared data need
STONITH to ensure data integrity
Errors found during check: config not valid
```

This shows up because the cluster has not been configured with a fencing agent, which is a piece of software used to isolate a host in the cluster framework when a fault is detected. Fencing configuration is covered in the next section.

Pacemaker Server Fault Isolation with Fencing

When a server or a software application develops a fault in a high-availability software framework, it is essential to isolate the affected component and remove it from operation in order to protect the integrity of the cluster as a whole, and to protect data hosted by the cluster from corruption. In shared-storage HA clusters, such as those running Lustre storage services, data corruption is most likely to occur when multiple services on different nodes try to access the same data storage concurrently. Each service assumes that it has exclusive access to the data, which leads to a risk that one service might overwrite information committed by another service on a different cluster node. There are some protections in place in Ldiskfs (and to a lesser extent ZFS) that minimize the risk of concurrent access, but HA software frameworks such as Pacemaker provide additional protections to further reduce exposure to this risk.

The mechanism for isolating a failed component is called fencing. Fencing is the means by which a node in a cluster is prevented from accessing the shared storage. This is usually achieved by forcing the failed node to power off. In Pacemaker, once a fault is detected, healthy nodes that are able to form a quorum create a new cluster configuration with the faulty node removed. The faulty node is then fenced, and any services that were running on the now isolated server are migrated to the surviving node or nodes.

In Pacemaker, the fencing mechanism most commonly used is STONITH, which stands for Shoot the Other Node in the Head. STONITH relies on healthy cluster nodes detecting a fault or failure in another node, and forcibly removing that faulted node from the cluster using a brute force mechanism such as power cycling the host.

Pacemaker has a set of software components called fencing agents that are used for this purpose. There are several such agents available, but most conform to the same basic principal of fault isolation through power control. All such agents rely on supporting infrastructure to facilitate power control, and each has its own specific requirements and parameters. Some agents, such as `fence_apc`, provide an interface to intelligent power distribution units (PDUs). These are vendor-specific interfaces, and provide a high level of reliability in the mechanism, since they require only that there is access to the PDU control interface.

There are also several vendor-specific BMC (Baseboard Management Controller) fence agents as well as a generic, hardware-agnostic IPMI agent; these are also reliable but do require that

the BMC is itself responsive in order to work. That is, the BMC must still have a supply of power.

There are several RPMs available that contain fencing agents. It is generally simplest to just install the superset RPM package, called fence-agents-all. One can also just supply the package name “fence-agents” to YUM and this will mark all of the fence agents in the RHEL OS repositories for installation.

To obtain a full list of the available agents installed on a host with the PCS software installed:

```
pcs stonith list
```

To get more detailed information about a specific agent:

```
pcs stonith describe <agent>
```

Fencing is notoriously difficult to configure correctly, as it is difficult to anticipate and test all of the potential failure modes. If the fencing agent does not exit cleanly and without reporting an error, then the fencing operation will be regarded by Pacemaker as failed, and the resources hosted by the failed node will not be migrated to a healthy cluster node. This is because the agent did not report success when isolating the affected node. Rather than risk compromising the integrity of any data associated with the resources by potentially running multiple services on both the healthy and unhealthy cluster nodes, Pacemaker will refuse to migrate resources until it can be sure that the faulty node has been isolated.

Configuring the IPMI Fence Agent For Pacemaker

Fencing is a complex topic and there are a number of parameters that can be set to control the behavior of fencing in a Pacemaker cluster. This guide is not a comprehensive reference on the topic, but provides an introduction to the basic mechanisms, illustrated by some examples.

The `fence_ipmilan` agent is one of the more versatile fencing agents available in the standard cluster software distribution. The typical required syntax for creating a fencing agent for a Pacemaker cluster is as follows:

```
pcs stonith create <resource name> fence_ipmilan \
  ipaddr="X.X.X.X" \
  [ lanplus=true ] \
  login="XXXX" passwd="XXXXX" \
  pcmk_host_list="<cluster node>"
```

The command must be run for each node in the cluster (each cluster node must have its own fence agent resource) but the commands can be run from a single node.

- `<resource name>` should be descriptive and is usually the cluster node name with the suffix "`ipmi`"
- `fence_ipmilan` is the name of the fence agent application
- `ipaddr` is the IP address of the BMC or equivalent target that will receive the IPMI request from the agent. It is *not* the IP address of the host that will be fenced.
- `lanplus` is optional but usually required by newer devices that support IPMI, as it represents an update to the protocol that improves the security of the connection. It should always be used unless the IPMI target does not support `lanplus`.
- `login` and `passwd` are use to supply the login credentials to the IPMI device
- `pcmk_host_list` is the name of the host registered in the Pacemaker configuration that will be controlled by this fencing agent. Some agents are able to control multiple nodes using a single agent, which is why the parameter has the suffix `_list`. However, for IPMI, there is a 1:1 correlation between the agent and the node to be fenced.

For example:

```
pcs stonith create rh7z-mds1-ipmi fence_ipmilan
ipaddr="10.10.10.111" \
  lanplus=true login="admin" passwd="newroot" \
  pcmk_host_list="rh7z-mds1.lfs.intl"

pcs stonith create rh7z-mds2-ipmi fence_ipmilan
ipaddr="10.10.10.112" \
  lanplus=true login="admin" passwd="newroot" \
  pcmk_host_list="rh7z-mds2.lfs.intl"
```

Be aware that the password will be recorded into the cluster's information base (CIB). This means that a user with suitable privileges on a cluster node will be able to retrieve the password of the IPMI user. If this is a concern, there is an option to supply the password via a script. Use the following command to review the available options:

```
pcs stonith describe fence_ipmilan
```

Creating Pacemaker Resources for Lustre Storage Services

Although Pacemaker is a commonly-used framework for providing high-availability for Lustre, there are currently no off-the-shelf resource agents available for Pacemaker to manage ZFS storage pools or Lustre file system OSDs that are based on ZFS, either from the ZFS on Linux project or from the Lustre project. This is at least in part a consequence of the fact that prior to the development of the ZFS OSD, Lustre's Ext-based Ldiskfs storage target could use the generic `ocf:heartbeat:Filesystem` resource agent, as no specific or special development was required.

Intel® has developed reference sample resource agents for Pacemaker and these have been successfully deployed in production at several sites. They are provided as-is, in the appendices of this document.

Lustre + ZFS Resource Agent Installation

To use the Lustre + ZFS resource agent, take the LustreZFS script from the appendix and install on each server that has Lustre ZFS OSDs. Install the script into the following directory:

```
/usr/lib/ocf/resource.d/heartbeat/
```

The file must have the name `LustreZFS`, owner root, group root and the permissions 755:

```
[root@rh7z-mds1 ~]# cd /usr/lib/ocf/resource.d/heartbeat
[root@rh7z-mds1 heartbeat]# ls -l LustreZFS
-rwxr-xr-x 1 root root 7398 Apr 15 00:24 LustreZFS
```

Lustre + ZFS Resource Agent Configuration for MGT and MDT0

To create the MGT resource in Pacemaker, run the following command from one of the metadata server cluster nodes:

```
pcs resource create lustreMGS ocf:heartbeat:LustreZFS \
  pool="<MGT pool name>" \
  volume="<MGT dataset>" \
  mountpoint="/lustre/mgt"
```

For example:

```
pcs resource create lustreMGS ocf:heartbeat:LustreZFS \
  pool="mgspool" \
  volume="mgt" \
  mountpoint="/lustre/mgt"
```

The syntax for creating a resource to manage an MDT is similar:

```
pcs resource create lustre-<fsname>MDT<n> ocf:heartbeat:LustreZFS \
  pool="<MDTn pool name>" \
  volume="<MDTn dataset>" \
  mountpoint="/lustre/<fsname>/mdt<n>"
```

For example:

```
pcs resource create lustre-demoMDT0 ocf:heartbeat:LustreZFS \
  pool="demo-mdt0pool" \
  volume="mdt0" \
  mountpoint="/lustre/demo/mdt0"
```

In theory, one could leave Pacemaker to determine the "best" way to manage the placement of the services n the cluster nodes, but in practice this can make the behavior of the cluster unpredictable. In addition, the Lustre clients will search for the Lustre services in a prescribed, static order, so it is best to try and ensure that the services are organized by Pacemaker in a manner that complies with the expectations of the Lustre clients.

What this means is that the Lustre services need to have constraints applied to them that establish preferences for where the services ought to run, assuming that the cluster is healthy.

The syntax is as follows:

```
pcs constraint location <resource> prefers <node>=<weight>
```

For the MGS and MDT0, the commands will look like this:

```
pcs constraint location lustreMGS \
  prefers rh7z-mds1.lfs.intl=20
pcs constraint location lustreMGS \
  prefers rh7z-mds2.lfs.intl=10

pcs constraint location lustre-demoMDT0 \
  prefers rh7z-mds2.lfs.intl=20
pcs constraint location lustre-demoMDT0 \
  prefers rh7z-mds1.lfs.intl=10
```

Notice that the weighting is higher for the preferred primary node and that the MGS will prefer to run on `rh7z-mds1` and the MDS for MDT0 will prefer to run on `rh7z-mds2`.

Lustre + ZFS Resource Agent Configuration for the OSTs

The configuration for the OSTs is very similar to that for the MGT and MDT, except that there are likely to be several OSTs running within a single OSS cluster pair:

```
pcs resource create lustre-<fsname>OST<n> ocf:heartbeat:LustreZFS \
  pool="<OSTn pool name>" \
  volume="<OSTn dataset>" \
  mountpoint="/lustre/<fsname>/ost<n>"

For example:
pcs resource create lustre-demoOST0 ocf:heartbeat:LustreZFS \
  pool="demo-ost0pool" \
  volume="ost0" \
  mountpoint="/lustre/demo/ost0"
```

Similar to the metadata server HA resource configuration, the Lustre OSTs need to have constraints applied to them that establish preferences for where the services ought to run, assuming that the cluster is healthy.

The syntax is as follows:

```
pcs constraint location <resource> prefers <node>=<weight>
```

For a cluster pair, `rh7z-oss1` and `rh7z-oss2`, with `OST0` and `OST1`, for example, the commands will look like this:

```
pcs constraint location lustre-demoOST0 \
  prefers rh7z-oss1.lfs.intl=20
pcs constraint location lustre-demoOST0 \
  prefers rh7z-oss2.lfs.intl=10

pcs constraint location lustre-demoOST1 \
  prefers rh7z-oss2.lfs.intl=20
pcs constraint location lustre-demoOST1 \
  prefers rh7z-oss1.lfs.intl=10
```

Again, similarly to the metadata server configuration, notice that the weighting is higher for the preferred primary node and that OST0 will prefer to run on `rh7z-oss1` and OST1 will prefer to run on `rh7z-oss2`.

Creating Additional Monitoring Resources In Pacemaker

The high-availability configuration can be extended to monitor the health of services running in the Pacemaker cluster. Resource agents have been developed to monitor the health of LNet and the status of Lustre services (MGS, MDS and OSS) running on the cluster nodes.

These additional agents are not specific to ZFS and can also be incorporated into a LDISKFS-based Lustre file system.

The resource agents make use of Pacemaker's resource clone feature to provide active/active monitoring across the cluster nodes. The cloned resources actively monitor the health of LNet and the Lustre services, updating specific variables in Pacemaker's configuration, called the Cluster Information Base (CIB). If a variable's value falls below a specific threshold, a Pacemaker constraint is triggered and all the resources that were running on the faulted node are migrated to a healthy node.

Detection of LNET Outage

The `healthLNET` script (available in the Appendix) provides two principal checks: it monitors the physical state of the network device that has been configured for LNet, and verifies connectivity to Lustre resources via the LNet protocol using `lctl ping`.

The following example demonstrates how to create a resource in Pacemaker to monitor LNet health:

```
# pcs resource create healthLNET ocf:pacemaker:healthLNET \
  dampen=5s multiplier=1000 lctl=true device=eth1 \
  host_list="10.10.130.1@tcp1 10.10.130.2@tcp1" \
  --clone
```

The `healthLNET` resource provides health status information about the LNet interface, but it does not, by itself, trigger a failover or fencing action. Instead, the monitoring resource is used to create a constraint on the Lustre services configured in the cluster. This constraint will force the services to migrate, if the monitoring resource reports a value that is lower than the acceptable threshold for the resource.

Each Lustre resource configured in the cluster pair requires a constraint to be applied.

The syntax for creating a location constraint against the LNet health monitor is as follows:

```
#pcs constraint location <resource name> \
  rule score=-INFINITY pingd lt 1 or not_defined pingd
```

Option	Description	OCF variable	Default Value
dampen	The time to wait (dampening) further changes occur.	OCF_RESKEY_dampen	5s
multiplier	The number by which to multiply the number of connected ping nodes by.	OCF_RESKEY_multiplier	1
attempts	Number of ping attempts, per host, before declaring it dead.	OCF_RESKEY_attempts	3
timeout	How long, in seconds, to wait before declaring a ping lost.	OCF_RESKEY_timeout	5
lctl	Option to enable lctl ping instead the linux ping.	OCF_RESKEY_lctl	True

device	Device used for the LNET network. We assume the same device across the cluster.	OCF_RESKEY_device	
host_list	List of IP or NID of the hosts to test. IP are used if the linux ping is selected. NID must be provided if the lctl ping is used.	OCF_RESKEY_host_list	

Detection of MDS/OSS/MGS services outages

The healthLUSTRE script (available in the Appendix) is used to verify the status of the Lustre services running on a host by monitoring the content of the file/proc/fs/lustre/health_check.

This file will report the following error conditions:

- LBUG (Lustre kernel bug, an irrecoverable error that halts execution of the kernel thread to avoid potential further corruption of the system state. The system must be rebooted to clear this state.
- Any I/O error on the target

The following example demonstrates the syntax for configuring the healthLUSTRE resource:

```
# pcs resource create healthLUSTRE \
  ocf:pacemaker:healthLUSTRE dampen=5s --clone
```

After the clone resource is created, create a constraint on each of the Lustre resources configured in the Pacemaker framework, such that an error reported by healthLUSTRE will trigger a failover of the resources. The syntax is as follows:

```
# pcs constraint location <resource name> \
  rule score=-INFINITY lustred lt 1 or not_defined lustred
```

Option	Description	OCF variable	Default Value
dampen	The time to wait (dampening) further changes occur	OCF_RESKEY_dampen	5s

Appendix A: RHEL / CentOS Kickstart Template

```
install
text
poweroff
lang en_US.UTF-8
keyboard us

# The next entry sets the network interface during installation
# Lustre servers will require statically allocated IP addresses
# Additional network interfaces to be configured as required
network --hostname=<name> --onboot=yes --device <dev> --bootproto dhcp --
noipv6 --nameserver 8.8.8.8 --gateway 0.0.0.0
# Disable the firewall
# If the firewall must be in place and Lustre is using TCP/IP
# for comms, then enable traffic on port 988.
firewall --disabled
# Disable SELinux
selinux --disabled

# Set the basic authentication algorithm and set initial root password
authconfig --enableshadow --passalgo=sha512
rootpw  --iscrypted <password hash; create with grub-crypt --sha-512>

timezone --utc America/New_York
bootloader --location=mbr --driveorder=sda --append="crashkernel=auto
console=ttyS0,115200 rd_NO_PLYMOUTH"
zerombr
clearpart --all --initlabel --drives=sda
autopart

%packages
@core
@base
# The following are optional but commonly used on Lustre servers
# Not normally required for Lustre clients
device-mapper-multipath
device-mapper-multipath-libs
%end

%pre
%end

%post
%end
```

Appendix B: Lustre ZFS Pacemaker Resource Agent

The following script can be used to manage ZFS and Lustre on shared storage. For each host, save this script as "LustreZFS" to the location:
`/usr/lib/ocf/resource.d/heartbeat/`

and set the script's permissions to 755.

```
#!/bin/sh
#
# License:      GNU General Public License (GPL)v2
# Description:  Manages ZFS and Lustre on a shared storage
# Written by:   Gabriele Paciucci
# Release Date: 01 June 2016
# Release Version: 0.98
# Copyright © 2016, Intel Corporation
#
# This program is free software; you can redistribute it and/or modify
# it under the terms and conditions of the GNU General Public License,
# version 2, as published by the Free Software Foundation.
#
# This program is distributed in the hope it will be useful, but
# WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
# General Public License for more details.
#
#
# usage: ./LustreZFS {start|stop|status|monitor|validate-all|meta-data}
#
#               OCF parameters are as follows
#               OCF_RESKEY_pool - the pool to import/export
#               OCF_RESKEY_volume - the volume to mount/umount
#               OCF_RESKEY_mountpoint - the mountpoint to use
#######################################################################
# Initialization:

: ${OCF_FUNCTIONS_DIR=${OCF_ROOT}/lib/heartbeat}
. ${OCF_FUNCTIONS_DIR}/ocf-shellfuncs

# Defaults

# Variables used by multiple methods


#######################################################################

# USAGE

usage() {
usage: $0 {start|stop|status|monitor|validate-all|meta-data}
}

# META-DATA

meta_data() {
cat <<END
```

```
<?xml version="1.0"?>
<!DOCTYPE resource-agent SYSTEM "ra-api-1.dtd">
<resource-agent name="LustreZFS">
<version>0.98</version>
<longdesc lang="en">
This script manages ZFS pools and Lustre volumes. The script is able to import and export ZFS
pools and mount/umount Lustre.
</longdesc>
<shortdesc lang="en">Lustre and ZFS management</shortdesc>

<parameters>

<parameter name="pool" unique="1" required="1">
<longdesc lang="en">
The name of the ZFS pool to manage.
</longdesc>
<shortdesc lang="en">ZFS pool name</shortdesc>
<content type="string" default="" />
</parameter>

<parameter name="volume" unique="1" required="1">
<longdesc lang="en">
The name of the volume created during the Lustre format on the ZFS pool.
</longdesc>
<shortdesc lang="en">Lustre volume name in the pool</shortdesc>
<content type="string" default="" />
</parameter>

<parameter name="mountpoint" unique="1" required="1">
<longdesc lang="en">
The mount point where the Lustre target will be mounted.
</longdesc>
<shortdesc lang="en">Mount point for Lustre</shortdesc>
<content type="string" default="" />
</parameter>


</parameters>

<actions>
<action name="start"   timeout="300s" />
<action name="stop"    timeout="300s" />
<action name="monitor" depth="0"  timeout="300s" interval="20s" />
<action name="validate-all"  timeout="30s" />
<action name="meta-data"  timeout="5s" />
</actions>
</resource-agent>
END
           exit $OCF_SUCCESS
}

# FUNCTIONS

zpool_is_imported () {
    zpool list -H "$OCF_RESKEY_pool" > /dev/null
}

lustre_is_mounted () {
# Verify if this is consistent
    grep -q "$OCF_RESKEY_mountpoint" /proc/mounts
}
```

```
zpool_import () {
    if ! zpool_is_imported; then
            ocf_log info "Starting to import $OCF_RESKEY_pool"

# We start with the assumption that the pool is "protected" by the ZFS's hostid mechanism
# We try first to import without forcing
# If the zpool import return an error, we try again to be sure
# If we got another error, we fencing the other node using stonith_adm -F <name node>
# At this point we can import the pool using the -f option
# The meanings of the options to import are as follows:
#   -f : import even if the pool is marked as imported
#   -o cachefile=none : the import should be temporary

# Try to import clean
                  if zpool import -o cachefile=none "$OCF_RESKEY_pool" ; then
                         ocf_log info "$OCF_RESKEY_pool imported successfully"
                                return $OCF_SUCCESS
                  else
                         ocf_log err "$OCF_RESKEY_pool import failed with this error $? we
are trying again after 5 seconds!"
                                sleep 5
                  fi
# Try to import clean again JIC

                                if zpool import -o cachefile=none "$OCF_RESKEY_pool" ; then
                                       ocf_log info "$OCF_RESKEY_pool imported
successfully"
                                              return $OCF_SUCCESS
                                else
                                       ocf_log err "$OCF_RESKEY_pool import failed with
this error $? we are fencing the other node"
                                              WhoAmI=$(crm_node -n)
# Valid only in a 2 node cluster
                                              WhoIsMyPair=$(crm_node -l| grep -v $WhoAmI|
cut -f2 -d" ")

                                              if stonith_admin -F $WhoIsMyPair ; then
                                                     ocf_log info "$WhoIsMyPair fenced
successfully"

# Try to import forced after stonith

                                                                    if zpool import -f -o
cachefile=none "$OCF_RESKEY_pool" ; then
                                                                           ocf_log info
"$OCF_RESKEY_pool imported successfully"
                                                                                  return
$OCF_SUCCESS
                                                                    else
                                                                           ocf_log err
"$OCF_RESKEY_pool FORCED import failed with this error $? "
                                                                                  return
$OCF_ERR_GENERIC
                                                                                  fi



                                              else
                                                     ocf_log err "$WhoIsMyPair fenced
failed with this error $? Please contact the support, safe import is not possible"
                                                                    return $OCF_ERR_GENERIC
```

```
                                                                fi

                                                                fi
                                                                fi
}

zpool_export () {
    if zpool_is_imported; then
            ocf_log info "Starting to export $OCF_RESKEY_pool"

# The meanings of the options to export are as follows:
#   -f : export in every case

                    if zpool export -f "$OCF_RESKEY_pool" ; then
                            ocf_log info "$OCF_RESKEY_pool exported successfully"
                                    return $OCF_SUCCESS
                    else
                            ocf_log err "$OCF_RESKEY_pool export failed"
                                    return $OCF_ERR_GENERIC
                                    fi
                                    fi
}


lustre_mount () {
    if ! lustre_is_mounted; then
            ocf_log info "Starting to mount $OCF_RESKEY_volume"

# The meanings of the options to export are as follows:
#

                    if mount -t lustre "$OCF_RESKEY_pool/$OCF_RESKEY_volume"
$OCF_RESKEY_mountpoint ; then

                            ocf_log info "$OCF_RESKEY_volume mounted successfully"
                                    return $OCF_SUCCESS
                    else
                            ocf_log err "$OCF_RESKEY_volume mount failed"
                                    return $OCF_ERR_GENERIC
                                    fi
                                    fi
}


lustre_umount () {

    if lustre_is_mounted; then
            ocf_log info "Starting to unmount $OCF_RESKEY_volume"

# The meanings of the options to export are as follows:
#  -f : force umount

                    if umount -f $OCF_RESKEY_mountpoint; then

                            ocf_log info "$OCF_RESKEY_volume unmounted successfully"
                                    return $OCF_SUCCESS
                    else
                            ocf_log err "$OCF_RESKEY_volume unmount failed"
                                    return $OCF_ERR_GENERIC
                                    fi
                                    fi
}
```

```
zpool_monitor () {

# If the pool is not imported, then we can't monitor its health
    if ! zpool_is_imported; then
            ocf_log warn "$OCF_RESKEY_pool not imported"
                    return $OCF_NOT_RUNNING
                    fi

##
## ATTENTION zpool status can hang in some conditions we disable at the moment this test
## in order to find a way to monitor a pool better
##
# Check the pool status
# Possible status:
#   DEGRADED
#   FAULTED
#   OFFLINE
#   ONLINE
#   REMOVED
#   UNAVAIL


#                   HEALTH=$(zpool list -H -o health "$OCF_RESKEY_pool")
#   case "$HEALTH" in
#                   ONLINE)                 #to debug ocf_log info "$OCF_RESKEY_pool is
$HEALTH"
#                           return $OCF_SUCCESS
#                           ;;
#                   DEGRADED)               ocf_log warn "$OCF_RESKEY_pool is $HEALTH"
#                           return $OCF_SUCCESS
#                           ;;
#                   FAULTED)                ocf_log err "$OCF_RESKEY_pool is $HEALTH"
#                           return $OCF_ERR_GENERIC
#                           ;;
#                   *)                      ocf_log err "$OCF_RESKEY_pool is $HEALTH"
#                           return $OCF_ERR_GENERIC
#                           ;;
#                   esac

                    return $OCF_SUCCESS

}


lustre_monitor () {

    if ! lustre_is_mounted; then
            ocf_log err "$OCF_RESKEY_volume is not mounted"
                    return $OCF_NOT_RUNNING

    else
# to debug ocf_log info "$OCF_RESKEY_volume is mounted"
            return $OCF_SUCCESS

                    fi

}
```

```
all_start () {

    zpool_import
    imp_success=$?
    if [ "$imp_success" != "$OCF_SUCCESS" ]; then
            ocf_log err "$OCF_RESKEY_pool can not be imported with this error: $imp_success"
            return $OCF_ERR_GENERIC
    else
            sleep 5
            lustre_mount
            mnt_success=$?

            if [ "$mnt_success" != "$OCF_SUCCESS" ]; then
                 ocf_log err "$OCF_RESKEY_volume can not be mounted with this error:
$mnt_success"
                 return $OCF_ERR_GENERIC
            fi
    fi

 return $OCF_SUCCESS


}



all_stop () {

    lustre_umount
        mnt_success=$?
        if [ "$mnt_success" != "$OCF_SUCCESS" ]; then
                ocf_log err "$OCF_RESKEY_volume can not be unmounted with this error:
$mnt_success"
                return $OCF_ERR_GENERIC
        else
                sleep 5
                zpool_export
                exp_success=$?

                if [ "$exp_success" != "$OCF_SUCCESS" ]; then
                ocf_log err "$OCF_RESKEY_volume can not be exported with this error:
$exp_success"
                return $OCF_ERR_GENERIC
                fi
        fi

 return $OCF_SUCCESS



}


all_monitor () {

##
## ATTENTION zpool status can hang in some conditions we disable at the moment this test
## in order to find a way to monitor a pool better
##
```

```
# if zpool_monitor return OCF_SUCCESS then execute lustre monitoring
    zpool_monitor
            zpool_result=$?

    case "$zpool_result" in
            $OCF_SUCCESS)           lustre_monitor
                    lustre_result=$?
                    if [ "$lustre_result" ==  $OCF_SUCCESS ]; then
                            return $OCF_SUCCESS
                                    fi
                                    return $OCF_NOT_RUNNING
                                    ;;
            $OCF_NOT_RUNNING)       # skip any additional tests and return
                    return $OCF_NOT_RUNNING
                    ;;
            $OCF_ERR_GENERIC)       # skip any additional tests and return the error
                    return $OCF_ERR_GENERIC
                    ;;
            *)                      ocf_log err "Unexpected result from the zpool_monitor
function"
                    return $OCF_ERR_GENERIC
                    ;;
            esac



}

validate () {

# Maybe we can implement some validation
    return $OCF_SUCCESS

}



case $1 in
meta-data)          meta_data;;
start)              all_start;;
stop)               all_stop;;
status|monitor)     all_monitor;;
validate-all)       validate;;
usage)              usage
exit $OCF_SUCCESS
;;
*)                  exit $OCF_ERR_UNIMPLEMENTED;;
esac

exit $?
```

Appendix C: LNet monitor Pacemaker Resource Agent

The following script can be used to manage ZFS and Lustre on shared storage. For each host, save this script as "`healthLNET`" to the location:
`/usr/lib/ocf/resource.d/pacemaker/`

and set the script's permissions to 755.

```
#!/bin/sh
#
#
#       LNet  OCF RA that utilizes the system ping
#


# License:      GNU General Public License (GPL)v2
# Description:  Manages ZFS and Lustre on a shared storage
# Written by:   Gabriele Paciucci
# Release Date: 01 June 2016
# Release Version: 0.98

# Copyright (c) 2009 Andrew Beekhof

# Copyright © 2016, Intel Corporation


#
# This program is free software; you can redistribute it and/or modify
# it under the terms of version 2 of the GNU General Public License as
# published by the Free Software Foundation.
#
# This program is distributed in the hope that it would be useful, but
# WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
#
# Further, this software is distributed without any warranty that it is
# free of the rightful claim of any third person regarding infringement
# or the like.  Any license provided herein, whether implied or
# otherwise, applies only to this software file.  Patent licenses, if
# any, provided herein do not apply to combinations of this program with
# other software, or any other product whatsoever.
#
# You should have received a copy of the GNU General Public License
# along with this program; if not, write the Free Software Foundation,
# Inc., 59 Temple Place - Suite 330, Boston MA 02111-1307, USA.
#

#######################################################################
# Initialization:

: ${OCF_FUNCTIONS=${OCF_ROOT}/resource.d/heartbeat/.ocf-shellfuncs}
. ${OCF_FUNCTIONS}
: ${__OCF_ACTION=$1}

#######################################################################

meta_data() {
       cat <<END
<?xml version="1.0"?>
<!DOCTYPE resource-agent SYSTEM "ra-api-1.dtd">
<resource-agent name="healthLNET">
<version>0.98</version>
```

```
<longdesc lang="en">
Every time the monitor action is run, this resource agent records (in the CIB) the current number
of lctl ping nodes the host can connect to.
</longdesc>
<shortdesc lang="en">LNet connectivity</shortdesc>

<parameters>

<parameter name="pidfile" unique="0">
<longdesc lang="en">PID file</longdesc>
<shortdesc lang="en">PID file</shortdesc>
<content type="string" default="$HA_VARRUN/ping-${OCF_RESOURCE_INSTANCE}" />
</parameter>

<parameter name="dampen" unique="0">
<longdesc lang="en">
The time to wait (dampening) further changes occur
</longdesc>
<shortdesc lang="en">Dampening interval</shortdesc>
<content type="integer" default="5s"/>
</parameter>

<parameter name="name" unique="0">
<longdesc lang="en">
The name of the attributes to set.  This is the name to be used in the constraints.
</longdesc>
<shortdesc lang="en">Attribute name</shortdesc>
<content type="string" default="pingd"/>
</parameter>

<parameter name="multiplier" unique="0">
<longdesc lang="en">
The number by which to multiply the number of connected ping nodes by
</longdesc>
<shortdesc lang="en">Value multiplier</shortdesc>
<content type="integer" default=""/>
</parameter>

<parameter name="host_list" unique="0" required="1">
<longdesc lang="en">
The list of ping nodes to count.
</longdesc>
<shortdesc lang="en">Host list</shortdesc>
<content type="string" default=""/>
</parameter>

<parameter name="attempts" unique="0">
<longdesc lang="en">
Number of ping attempts, per host, before declaring it dead
</longdesc>
<shortdesc lang="en">no. of ping attempts</shortdesc>
<content type="integer" default="2"/>
</parameter>

<parameter name="timeout" unique="0">
<longdesc lang="en">
How long, in seconds, to wait before declaring a ping lost
</longdesc>
<shortdesc lang="en">ping timeout in seconds</shortdesc>
<content type="integer" default="2"/>
</parameter>

<parameter name="lctl" unique="0">
<longdesc lang="en">
Option to enable lctl ping. The default is true
</longdesc>
<shortdesc lang="en">Extra Options</shortdesc>
<content type="string" default="true"/>
</parameter>

<parameter name="device" unique="0">
```

```
<longdesc lang="en">
Device used for the LNET network. We assume the same device accross the cluster
</longdesc>
<shortdesc lang="en">LNET device</shortdesc>
<content type="string" default=""/>
</parameter>


<parameter name="options" unique="0">
<longdesc lang="en">
A catch all for any other options that need to be passed to ping.
</longdesc>
<shortdesc lang="en">Extra Options</shortdesc>
<content type="string" default=""/>
</parameter>

<parameter name="failure_score" unique="0">
<longdesc lang="en">
Resource is failed if the score is less than failure_score.
Default never fails.
</longdesc>
<shortdesc lang="en">failure_score</shortdesc>
<content type="integer" default=""/>
</parameter>

<parameter name="debug" unique="0">
<longdesc lang="en">
Enables to use default attrd_updater verbose logging on every call.
</longdesc>
<shortdesc lang="en">Verbose logging</shortdesc>
<content type="string" default="false"/>
</parameter>

</parameters>

<actions>
<action name="start"   timeout="300s" />
<action name="stop"    timeout="300s" />
<action name="reload"  timeout="300s" />
<action name="monitor" depth="0"  timeout="300s" interval="20s"/>
<action name="meta-data"  timeout="5" />
<action name="validate-all"  timeout="30" />
</actions>
</resource-agent>
END
}

#######################################################################

ping_conditional_log() {
    level=$1; shift
    if [ ${OCF_RESKEY_debug} = "true" ]; then
       ocf_log $level "$*"
    fi
}

ping_usage() {
        cat <<END
usage: $0 {start|stop|monitor|migrate_to|migrate_from|validate-all|meta-data}

Expects to have a fully populated OCF RA-compliant environment set.
END
}

ping_start() {
    ping_monitor
    if [ $? =  $OCF_SUCCESS ]; then
       return $OCF_SUCCESS
    fi
    touch ${OCF_RESKEY_pidfile}
    ping_update
```

```
}

ping_stop() {

        rm -f ${OCF_RESKEY_pidfile}

    attrd_updater -D -n $OCF_RESKEY_name -d $OCF_RESKEY_dampen $attrd_options

    return $OCF_SUCCESS
}

ping_monitor() {
    if [ -f ${OCF_RESKEY_pidfile} ]; then
        ping_update
        if [ $? -eq 0 ]; then
            return $OCF_SUCCESS
        fi
        return $OCF_ERR_GENERIC
    fi
    return $OCF_NOT_RUNNING
}

ping_validate() {
    # Is the state directory writable?
    state_dir=`dirname "$OCF_RESKEY_pidfile"`
    touch "$state_dir/$$"
    if [ $? != 0 ]; then
        ocf_log err "Invalid location for 'state': $state_dir is not writable"
        return $OCF_ERR_ARGS
    fi
    rm "$state_dir/$$"

# Pidfile better be an absolute path
    case $OCF_RESKEY_pidfile in
        /*) ;;
        *) ocf_log warn "You should use an absolute path for pidfile not: $OCF_RESKEY_pidfile" ;;
    esac

# Check the host list
    if [ "x" = "x$OCF_RESKEY_host_list" ]; then
        ocf_log err "Empty host_list.  Please specify some nodes to ping"
        exit $OCF_ERR_CONFIGURED
    fi

    check_binary ping

    return $OCF_SUCCESS
}

lctl_check() {
    active=0
    for host in $OCF_RESKEY_host_list; do
        lctl_exe="lctl ping"

        lctl_out=`$lctl_exe $host $OCF_RESKEY_timeout 2>&1`; rc=$?
# debug
#ocf_log info "$lctl_exe $host $OCF_RESKEY_timeout"

        case $rc in
            0) active=`expr $active + 1`;;
            1) ping_conditional_log warn "$host is inactive: $lctl_out";;
            *) ocf_log err "Unexpected result for '$lctl_exe $host $OCF_RESKEY_timeout' $rc:
$p_out";;
        esac
    done
    return $active


}
```

```
ping_check() {
    active=0
    for host in $OCF_RESKEY_host_list; do
       p_exe=ping

       case `uname` in
           Linux) p_args="-n -q -W $OCF_RESKEY_timeout -c $OCF_RESKEY_attempts";;
           Darwin) p_args="-n -q -t $OCF_RESKEY_timeout -c $OCF_RESKEY_attempts -o";;
           *) ocf_log err "Unknown host type: `uname`"; exit $OCF_ERR_INSTALLED;;
       esac

       case $host in
           *:*) p_exe=ping6
       esac

       p_out=`$p_exe $p_args $OCF_RESKEY_options $host 2>&1`; rc=$?

       case $rc in
           0) active=`expr $active + 1`;;
           1) ping_conditional_log warn "$host is inactive: $p_out";;
           *) ocf_log err "Unexpected result for '$p_exe $p_args $OCF_RESKEY_options $host' $rc:
$p_out";;
       esac
    done
    return $active
}

ping_update() {

# first I'm testing if I have the physical link up. If not I give up without any additional
tests.
# but first we need to find which is the device we are using on the local host, this is very
difficult with NID
# we need to add another variable

CARRIER=/sys/class/net/$OCF_RESKEY_device/carrier
OPERSTATE=/sys/class/net/$OCF_RESKEY_device/operstate

CAR_STAT=$(cat $CARRIER)
OPER_STAT=$(cat $OPERSTATE)

# debug
# ocf_log info "$CAR_STAT - $OPER_STAT"


if [ "$CAR_STAT" == "1" ] && [ "$OPER_STAT" == "up" ]; then
       if [ ${OCF_RESKEY_lctl} = "true" ]; then
              lctl_check
              active=$?
       else
              ping_check
              active=$?
       fi
else
       active=0
fi

# debug
#ocf_log info "$active"

    score=`expr $active \* $OCF_RESKEY_multiplier`
    attrd_updater -n $OCF_RESKEY_name -v $score -d $OCF_RESKEY_dampen $attrd_options
    rc=$?
    case $rc in
       0) ping_conditional_log debug "Updated $OCF_RESKEY_name = $score" ;;
       *) ocf_log warn "Could not update $OCF_RESKEY_name = $score: rc=$rc";;
    esac
    if [ $rc -ne 0 ]; then
        return $rc
    fi
```

```
    if [ -n "$OCF_RESKEY_failure_score" -a "$score" -lt "$OCF_RESKEY_failure_score" ]; then
        ocf_log warn "$OCF_RESKEY_name is less than failure_score($OCF_RESKEY_failure_score)"
        return 1
    fi
    return 0
}

: ${OCF_RESKEY_name:="pingd"}
: ${OCF_RESKEY_dampen:="5s"}
: ${OCF_RESKEY_attempts:="3"}
: ${OCF_RESKEY_multiplier:="1"}
: ${OCF_RESKEY_debug:="false"}
: ${OCF_RESKEY_lctl:="true"}
#: ${OCF_RESKEY_device:="eth1"}
: ${OCF_RESKEY_failure_score:="0"}

: ${OCF_RESKEY_CRM_meta_timeout:="20000"}
: ${OCF_RESKEY_CRM_meta_globally_unique:="true"}

integer=`echo ${OCF_RESKEY_timeout} | egrep -o '[0-9]*'`
case ${OCF_RESKEY_timeout} in
    *[0-9]ms|*[0-9]msec) OCF_RESKEY_timeout=`expr $integer / 1000`;;
    *[0-9]m|*[0-9]min) OCF_RESKEY_timeout=`expr $integer \* 60`;;
    *[0-9]h|*[0-9]hr)  OCF_RESKEY_timeout=`expr $integer \* 60 \* 60`;;
    *) OCF_RESKEY_timeout=$integer;;
esac

if [ -z ${OCF_RESKEY_timeout} ]; then
    if [ x"$OCF_RESKEY_host_list" != x ]; then
        host_count=`echo $OCF_RESKEY_host_list | awk '{print NF}'`
        OCF_RESKEY_timeout=`expr $OCF_RESKEY_CRM_meta_timeout / $host_count /
$OCF_RESKEY_attempts`
        OCF_RESKEY_timeout=`expr $OCF_RESKEY_timeout / 1100` # Convert to seconds and finish 10%
early
    else
        OCF_RESKEY_timeout=5
    fi
fi

if [ ${OCF_RESKEY_timeout} -lt 1 ]; then
    OCF_RESKEY_timeout=5
elif [ ${OCF_RESKEY_timeout} -gt 1000 ]; then
    # ping actually complains if this value is too high, 5 minutes is plenty
    OCF_RESKEY_timeout=300
fi

if [ ${OCF_RESKEY_CRM_meta_globally_unique} = "false" ]; then
    : ${OCF_RESKEY_pidfile:="$HA_VARRUN/ping-${OCF_RESKEY_name}"}
else
    : ${OCF_RESKEY_pidfile:="$HA_VARRUN/ping-${OCF_RESOURCE_INSTANCE}"}
fi

attrd_options='-q'
if ocf_is_true ${OCF_RESKEY_debug} ; then
    attrd_options=''
fi

# Check the debug option
    case "${OCF_RESKEY_debug}" in
        true|True|TRUE|1)    OCF_RESKEY_debug=true;;
        false|False|FALSE|0) OCF_RESKEY_debug=false;;
        *)
            ocf_log warn "Value for 'debug' is incorrect. Please specify 'true' or 'false' not:
${OCF_RESKEY_debug}"
            OCF_RESKEY_debug=false
            ;;
    esac

case $__OCF_ACTION in
meta-data)     meta_data
```

```
                exit $OCF_SUCCESS
                ;;
start)          ping_start;;
stop)           ping_stop;;
monitor)        ping_monitor;;
reload)         ping_start;;
validate-all)   ping_validate;;
usage|help)     ping_usage
                exit $OCF_SUCCESS
                ;;
*)              ping_usage
                exit $OCF_ERR_UNIMPLEMENTED
                ;;
esac
exit $?
```

Appendix D: Lustre services monitor Pacemaker Resource Agent

The following script can be used to manage ZFS and Lustre on shared storage. For each host, save this script as "`healthLUSTRE`" to the location:
`/usr/lib/ocf/resource.d/pacemaker/`

and set the script's permissions to 755.

```
#!/bin/sh
#
#
#       HealthLUSTRE OCF RA that utilizes the lustre /proc/fs/lustre/health_check
#

# License:      GNU General Public License (GPL)v2
# Description:  Manages ZFS and Lustre on a shared storage
# Written by:   Gabriele Paciucci
# Release Date: 01 June 2016
# Release Version: 0.97

# Copyright (c) 2009 Andrew Beekhof

# Copyright © 2016, Intel Corporation


#
# This program is free software; you can redistribute it and/or modify
# it under the terms of version 2 of the GNU General Public License as
# published by the Free Software Foundation.
#
# This program is distributed in the hope that it would be useful, but
# WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
#
# Further, this software is distributed without any warranty that it is
# free of the rightful claim of any third person regarding infringement
# or the like.  Any license provided herein, whether implied or
# otherwise, applies only to this software file.  Patent licenses, if
# any, provided herein do not apply to combinations of this program with
# other software, or any other product whatsoever.
#
# You should have received a copy of the GNU General Public License
# along with this program; if not, write the Free Software Foundation,
# Inc., 59 Temple Place - Suite 330, Boston MA 02111-1307, USA.
#

#######################################################################
# Initialization:

: ${OCF_FUNCTIONS=${OCF_ROOT}/resource.d/heartbeat/.ocf-shellfuncs}
. ${OCF_FUNCTIONS}
: ${__OCF_ACTION=$1}

#######################################################################

meta_data() {
        cat <<END
<?xml version="1.0"?>
<!DOCTYPE resource-agent SYSTEM "ra-api-1.dtd">
<resource-agent name="healthLUSTRE">
<version>1.0</version>

<longdesc lang="en">
```

```
Every time the monitor action is run, this resource agent records (in the CIB) the current number
of healthy lustre server
</longdesc>
<shortdesc lang="en">lustre servers healthy</shortdesc>

<parameters>

<parameter name="pidfile" unique="0">
<longdesc lang="en">PID file</longdesc>
<shortdesc lang="en">PID file</shortdesc>
<content type="string" default="$HA_VARRUN/healthLUSTRE-${OCF_RESOURCE_INSTANCE}" />
</parameter>

<parameter name="dampen" unique="0">
<longdesc lang="en">
The time to wait (dampening) further changes occur
</longdesc>
<shortdesc lang="en">Dampening interval</shortdesc>
<content type="integer" default="5s"/>
</parameter>


<parameter name="name" unique="0">
<longdesc lang="en">
The name of the attributes to set.  This is the name to be used in the constraints.
</longdesc>
<shortdesc lang="en">Attribute name</shortdesc>
<content type="string" default="lustred"/>
</parameter>



<parameter name="debug" unique="0">
<longdesc lang="en">
Enables to use default attrd_updater verbose logging on every call.
</longdesc>
<shortdesc lang="en">Verbose logging</shortdesc>
<content type="string" default="false"/>
</parameter>

</parameters>

<actions>
<action name="start"   timeout="60" />
<action name="stop"    timeout="20" />
<action name="reload"  timeout="100" />
<action name="monitor" depth="0"  timeout="60" interval="10"/>
<action name="meta-data"  timeout="5" />
<action name="validate-all"  timeout="30" />
</actions>
</resource-agent>
END
}

#######################################################################

lustre_conditional_log() {
    level=$1; shift
    if [ ${OCF_RESKEY_debug} = "true" ]; then
       ocf_log $level "$*"
    fi
}

lustre_usage() {
       cat <<END
usage: $0 {start|stop|monitor|migrate_to|migrate_from|validate-all|meta-data}

Expects to have a fully populated OCF RA-compliant environment set.
END
}
```

```
lustre_start() {
    lustre_monitor
    if [ $? =  $OCF_SUCCESS ]; then
       return $OCF_SUCCESS
    fi
    touch ${OCF_RESKEY_pidfile}
    lustre_update
}

lustre_stop() {

       rm -f ${OCF_RESKEY_pidfile}

    attrd_updater -D -n $OCF_RESKEY_name -d $OCF_RESKEY_dampen $attrd_options

    return $OCF_SUCCESS
}

lustre_monitor() {
    if [ -f ${OCF_RESKEY_pidfile} ]; then
       lustre_update
        if [ $? -eq 0 ]; then
            return $OCF_SUCCESS
        fi
        return $OCF_ERR_GENERIC
    fi
    return $OCF_NOT_RUNNING
}

# we worked on this later
#ping_validate() {
    # Is the state directory writable?
#    state_dir=`dirname "$OCF_RESKEY_pidfile"`
#    touch "$state_dir/$$"
#    if [ $? != 0 ]; then
#      ocf_log err "Invalid location for 'state': $state_dir is not writable"
#      return $OCF_ERR_ARGS
#    fi
#    rm "$state_dir/$$"

# Pidfile better be an absolute path
#    case $OCF_RESKEY_pidfile in
#      /*) ;;
#      *) ocf_log warn "You should use an absolute path for pidfile not: $OCF_RESKEY_pidfile" ;;
#    esac

# Check the host list
#    if [ "x" = "x$OCF_RESKEY_host_list" ]; then
#
#      ocf_log err "Empty host_list.  Please specify some nodes to ping"
#      exit $OCF_ERR_CONFIGURED
#    fi

#    check_binary ping

#    return $OCF_SUCCESS
#}



lustre_check() {

       active=0

#      added head -1 due the LU-7486
       l_out=`cat /proc/fs/lustre/health_check | head -1 |grep -w healthy 2>&1`; rc=$?

       case $rc in
           0) active=`expr $active + 1`;;
           1) lustre_conditional_log warn "Lustre is not healthy: $l_out";;
           *) ocf_log err "Unexpected result for '/proc/fs/lustre/health_check' $rc: $l_out";;
```

```
       esac
    return $active
}

lustre_update() {

       lustre_check
       active=$?

    attrd_updater -n $OCF_RESKEY_name -v $active -d $OCF_RESKEY_dampen $attrd_options
    rc=$?
    case $rc in
       0) lustre_conditional_log debug "Updated $OCF_RESKEY_name = $active" ;;
       *) ocf_log warn "Could not update $OCF_RESKEY_name = $active: rc=$rc";;
    esac
    if [ $rc -ne 0 ]; then
        return $rc
    fi
    return 0
}

: ${OCF_RESKEY_name:="lustred"}
: ${OCF_RESKEY_dampen:="5s"}
: ${OCF_RESKEY_attempts:="3"}
: ${OCF_RESKEY_debug:="false"}

: ${OCF_RESKEY_CRM_meta_timeout:="20000"}
: ${OCF_RESKEY_CRM_meta_globally_unique:="true"}


# I don't think we need to care about timeout

#integer=`echo ${OCF_RESKEY_timeout} | egrep -o '[0-9]*'`
#case ${OCF_RESKEY_timeout} in
#    *[0-9]ms|*[0-9]msec) OCF_RESKEY_timeout=`expr $integer / 1000`;;
#    *[0-9]m|*[0-9]min) OCF_RESKEY_timeout=`expr $integer \* 60`;;
#    *[0-9]h|*[0-9]hr)  OCF_RESKEY_timeout=`expr $integer \* 60 \* 60`;;
#    *) OCF_RESKEY_timeout=$integer;;
#esac

#if [ -z ${OCF_RESKEY_timeout} ]; then
#    if [ x"$OCF_RESKEY_host_list" != x ]; then
#       host_count=`echo $OCF_RESKEY_host_list | awk '{print NF}'`
#       OCF_RESKEY_timeout=`expr $OCF_RESKEY_CRM_meta_timeout / $host_count /
$OCF_RESKEY_attempts`
#       OCF_RESKEY_timeout=`expr $OCF_RESKEY_timeout / 1100` # Convert to seconds and finish 10%
early
#    else
#       OCF_RESKEY_timeout=5
#    fi
#fi

#if [ ${OCF_RESKEY_timeout} -lt 1 ]; then
#    OCF_RESKEY_timeout=5
#elif [ ${OCF_RESKEY_timeout} -gt 1000 ]; then
#    # ping actually complains if this value is too high, 5 minutes is plenty
#    OCF_RESKEY_timeout=300
#fi

if [ ${OCF_RESKEY_CRM_meta_globally_unique} = "false" ]; then
    : ${OCF_RESKEY_pidfile:="$HA_VARRUN/healthLUSTRE-${OCF_RESKEY_name}"}
else
    : ${OCF_RESKEY_pidfile:="$HA_VARRUN/healthLUSTRE-${OCF_RESOURCE_INSTANCE}"}
fi

attrd_options='-q'
if ocf_is_true ${OCF_RESKEY_debug} ; then
    attrd_options=''
fi

# Check the debug option
```

```
    case "${OCF_RESKEY_debug}" in
       true|True|TRUE|1)    OCF_RESKEY_debug=true;;
       false|False|FALSE|0) OCF_RESKEY_debug=false;;
       *)
            ocf_log warn "Value for 'debug' is incorrect. Please specify 'true' or 'false' not:
${OCF_RESKEY_debug}"
           OCF_RESKEY_debug=false
           ;;
    esac

case $__OCF_ACTION in
meta-data)     meta_data
               exit $OCF_SUCCESS
               ;;
start)         lustre_start;;
stop)          lustre_stop;;
monitor)       lustre_monitor;;
reload)        lustre_start;;
validate-all)  lustre_usage
               exit $OCF_SUCCESS
               ;;
usage|help)    lustre_usage
               exit $OCF_SUCCESS
               ;;
*)             lustre_usage
               exit $OCF_ERR_UNIMPLEMENTED
               ;;
esac
exit $?
```

References

High Performance Parallel I/O, Chapman and Hall/CRC Press, Prabhat, Koziol (Editors), ISBN: 978-1-4665-8234-7

Lustre Operations Manual: https://wiki.hpdd.intel.com/display/PUB/Documentation

Pacemaker project: http://clusterlabs.org/

Corosync project: http://corosync.github.io/corosync/

Linux-HA project: http://linux-ha.org/

https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Cluster_Administration/s1-config-rrp-cli-CA.html

https://access.redhat.com/solutions/162193

https://en.wikipedia.org/wiki/Dynamic_Kernel_Module_Support