# Administrator's Guide

## Scyld ClusterWare Release 7.3.0-730g0000

## January 20, 2017

**Administrator's Guide: Scyld ClusterWare Release 7.3.0-730g0000; January 20, 2017**

Revised Edition

Published January 20, 2017



# Table of Contents

# Preface

Welcome to the Scyld ClusterWare HPC Administrator's Guide. It is written for use by Scyld ClusterWare administrators and advanced users. This document covers cluster configuration, maintenance, and optimization. As is typical for any Linux-based system, the administrator must have root privileges to perform the administrative tasks described in this document.

The beginning of this guide describes the Scyld ClusterWare system architecture and design; it is critical to understand this information in order to properly configure and administer the cluster. The guide then provides specific information about tools and methods for setting up and maintaining the cluster, the cluster boot process, ways to control cluster usage, methods for batching jobs and controlling the job queue, how load balancing is handled in the cluster, and optional tools that can be useful in administrating your cluster. Finally, the an appendix covers the important files and directories that pertain to operation of Scyld ClusterWare.

This guide is written with the assumption that the administrator has a background in a Unix or Linux operating environment; therefore, the document does not cover basic Linux system administration. If you do not have sufficient knowledge for using or administering a Linux system, we recommend that you first consult *Linux in a Nutshell* and other useful books published by *O'Reilly and Associates*[1].

When appropriate, this document refers the reader to other parts of the Scyld documentation set for more detailed explanations of the topic at hand. For information on the initial installation of Scyld ClusterWare, refer to the *Installation Guide*, which provides explicit detail on setting up the master node and booting the compute nodes. For administrators who are new to the ClusterWare concept, we recommend reading the *User's Guide* first, as it introduces ClusterWare computing concepts.

## Feedback

We welcome any reports on errors or difficulties that you may find. We also would like your suggestions on improving this document. Please direct all comments and problems to support@penguincomputing.com.

When writing your email, please be as specific as possible, especially with errors in the text. Please include the chapter and section information. Also, please mention in which version of the manual you found the error. This version is *Scyld ClusterWare HPC, Revised Edition*, published January 20, 2017.

## Notes

1. http://www.oreilly.com

# Chapter 1. Scyld ClusterWare Design Overview

This chapter discusses the design behind Scyld ClusterWare, beginning with a high-level description of the system architecture for the cluster as a whole, including the hardware context, network topologies, data flows, software context, and system level files. From there, the discussion moves into a technical description that includes the compute node boot procedure, the process migration technology, compute node categories and states, and miscellaneous components. Finally, the discussion focuses on the ClusterWare software components, including tools, daemons, clients, and utilities.

As mentioned in the preface, this document assumes a certain level of knowledge from the reader and therefore, it does not cover any system design decisions related to a basic Linux system. In addition, it is assumed the reader has a general understanding of Linux clustering concepts and how the second generation Scyld ClusterWare system differs from the traditional Beowulf. For more information on these topics, see the *User's Guide*.

## System Architecture

Scyld ClusterWare provides a software infrastructure designed specifically to streamline the process of configuring, administering, running, and maintaining commercial production Linux cluster systems. Scyld ClusterWare installs on top of a standard Linux distribution on a single node, allowing that node to function as the control point or "master node" for the entire cluster of "compute nodes".

This section discusses the Scyld ClusterWare hardware context, network topologies, system data flow, system software context, and system level files.

### System Hardware Context

A Scyld cluster has three primary components:

- The master node
- Compute nodes
- The cluster private network interface

These components are illustrated in the following block diagram. The remaining element in the diagram is the public/building network interface connected to the master node. This network connection is not required for the cluster to operate properly, and may not even be connected (for example, for security reasons).

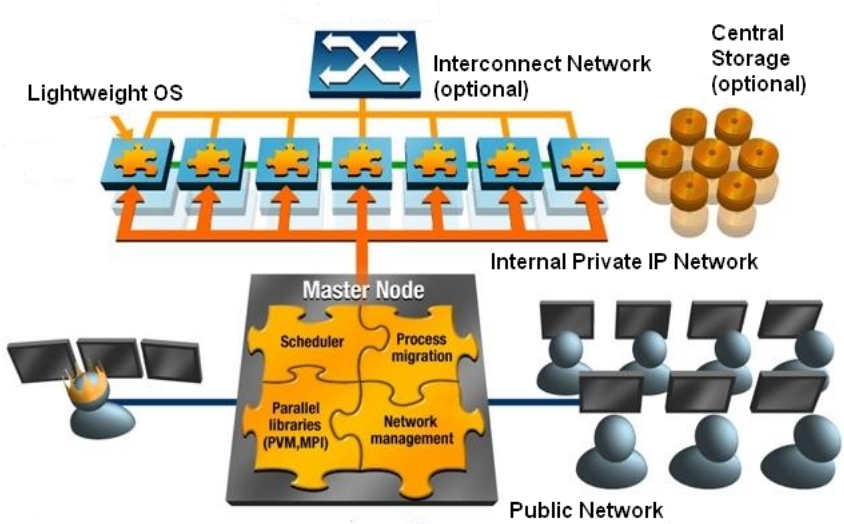


**Figure 1-1. Cluster Configuration**

The master node and compute nodes have different roles in Scyld ClusterWare, and thus they have different hardware requirements. The master node is the central administration console for the cluster; it is the machine that all users of the cluster log into for starting their jobs. The master node is responsible for sending these jobs out to the appropriate compute node(s) for execution. The master node also performs all the standard tasks of a Linux machine, such as queuing print jobs or running shells for individual users.

## Master Node

Given the role of the master node, it is easy to see why its hardware closely resembles that of a standard Linux machine. The master node will typically have the standard human user interface devices such as a monitor, keyboard, and mouse. It may have a fast 3D video card, depending on the cluster's application.

The master is usually equipped with two network interface cards (NICs). One NIC connects the master to the cluster's compute nodes over the private cluster network, and the other NIC connects the master to the outside world.

The master should be equipped with enough hard disk space to satisfy the demands of its users and the applications it must execute. The Linux operating system and Scyld ClusterWare together use about 7 GB of disk space. We recommend at least a 20 GB hard disk for the master node.

The master node should contain a minimum of 2 GB of RAM, or enough RAM to avoid swap during normal operations; a minimum of 4 GB is recommended. Having to swap programs to disk will degrade performance significantly, and RAM is relatively cheap.

Any network attached storage should be connected to both the private cluster network and the public network through separate interfaces.

In addition, if you plan to create boot CDs for your compute nodes, the master node requires a CD-RW or writeable DVD drive.

## Compute Nodes

In contrast to the master node, the compute nodes are single-purpose machines. Their role is to run the jobs sent to them by the master node. If the cluster is viewed as a single large-scale parallel computer, then the compute nodes are its CPU and

memory resources. They don't have any login capabilities, other than optionally accepting ssh connections from the master node, and aren't running many of the daemons typically found on a standard Linux box. These nodes don't need a monitor, keyboard, or mouse.

Video cards aren't required for compute nodes either (but may be required by the BIOS). However, having an inexpensive video card installed may prove cost effective when debugging hardware problems.

To facilitate debugging of hardware and software configuration problems on compute nodes, Scyld ClusterWare provides forwarding of all kernel log messages to the master's log, and all messages generated while booting a compute node are also forwarded to the master node. Another hardware debug solution is to use a serial port connection back to the master node from the compute nodes. The kernel command line options for a compute node can be configured to display all boot information to the serial port. See the Section called *Compute node command-line options* in Chapter 3 for details about the *console*= configuration setting.

Compute node RAM requirements are dependent upon the needs of the jobs that execute on the node. Compute node physical memory is shared between its RAM-based root filesystem (*rootfs*) and the runtime memory needs of user applications and the kernel itself. As more space is consumed by the root filesystem for files, less physical memory is available to applications' virtual memory and kernel physical memory, a shortage of which leads to Out-Of-Memory (*OOM*) events that result in application failure(s) and potentially total node failure.

Various remedies exist if the workloads fill the root filesystem or trigger Out-Of-Memory events, including adding RAM to the node and/or adding a local harddrive, which can be configured to add adequate swap space (which expands the available virtual memory capacity) and/or to add local filesystems (to reduce the demands on the RAM-based root filesystem). Even if local swap space is available and sufficient to avoid OOM events, optimal performance will only be achieved when there is sufficient physical memory to avoid swapping in the first place. See the Section called *Compute Node Failure* in Chapter 9 for a broader discussion of node failures, and the Section called *Compute node command-line options* in Chapter 3 for a discussion of the *rootfs_size*= configuration setting that limits the maximum size of the root filesystem.

A harddrive is not a required component for a compute node. If employed, we recommend using such local storage for data that can be easily re-created, such as swap space, scratch storage, or local copies of globally-available data.

If the compute nodes do not support PXE boot, a bootable CD-ROM drive is required.

## Network Topologies

For many applications that will be run on Scyld ClusterWare, an inexpensive Ethernet network is all that is needed. Other applications might require multiple networks to obtain the best performance; these applications generally fall into two categories, "message intensive" and "server intensive". The following sections describe a minimal network configuration, a performance network for "message intensive" applications, and a server network for "server intensive" applications.

### Minimal Network Configuration

Scyld ClusterWare requires that at least one IP network be installed to enable master and compute node communications. This network can range in speed from 10 Mbps (Fast Ethernet) to over 1 Gbps, depending on cost and performance requirements.

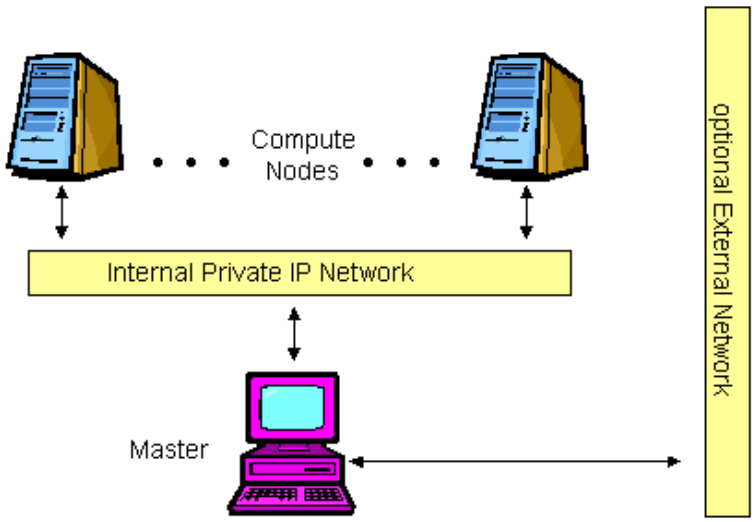


**Figure 1-2. Minimal Network Configuration**

## Performance Network Configuration

The performance network configuration is intended for applications that can benefit from the low message latency of proprietary networks like Infiniband, TOE Ethernet, or RDMA Ethernet. These networks can optionally run without the overhead of an IP stack with direct memory-to-memory messaging. Here the lower bandwidth requirements of the Scyld software can be served by a standard IP network, freeing the other network from any OS-related overhead completely.

It should be noted that these high performance interfaces may also run an IP stack, in which case they may also be used in the other configurations as well.

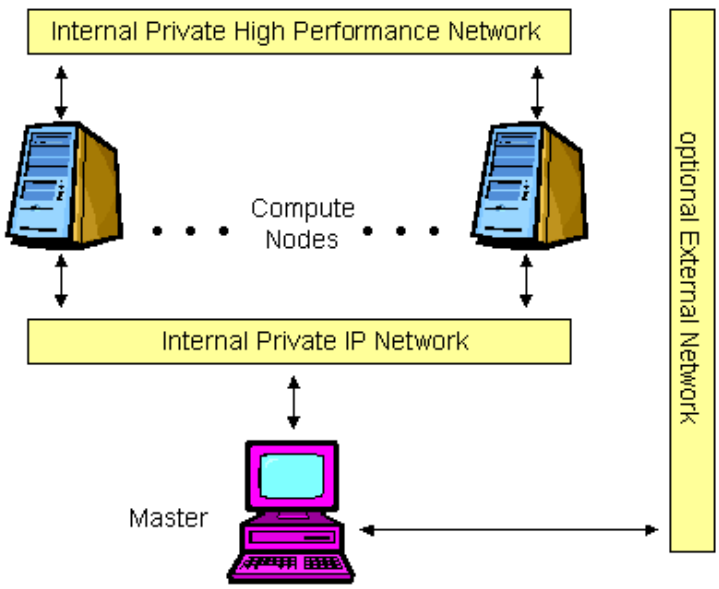


**Figure 1-3. Performance Network Configuration**

### Server Network Configuration

The server network configuration is intended for web, database, or application servers. In this configuration, each compute node has multiple network interfaces, one for the private control network and one or more for the external public networks.

The Scyld ClusterWare security model is well-suited for this configuration. Even though the compute nodes have a public network interface, there is no way to log into them. There is no `/etc/passwd` file or other configuration files to hack. There are no shells on the compute nodes to execute user programs. The only open ports on the public network interface are the ones your specific application opened.

To maintain this level of security, you may wish to have the master node on the internal private network only. The setup for this type of configuration is not described in this document, because it is very dependent on your target deployment. Contact Scyld's technical support for help with a server network configuration.

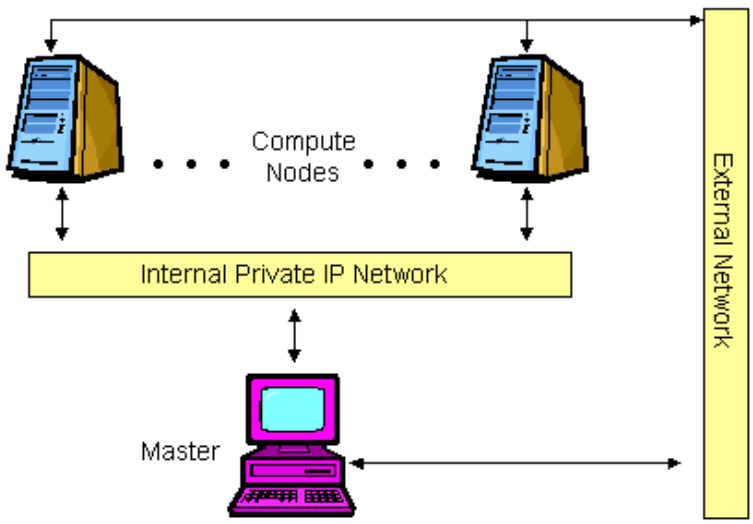


**Figure 1-4. Server Network Configuration**

## System Data Flow

The following data flow diagram shows the primary messages sent over the private cluster network between the master node and compute nodes in a Scyld cluster. Data flows in three ways:

- From the master node to the compute nodes
- From the compute nodes to the master node
- From the compute nodes to other compute nodes

The job control commands and cluster admin commands shown in the data flow diagram represent inputs to the master from users and administrators.

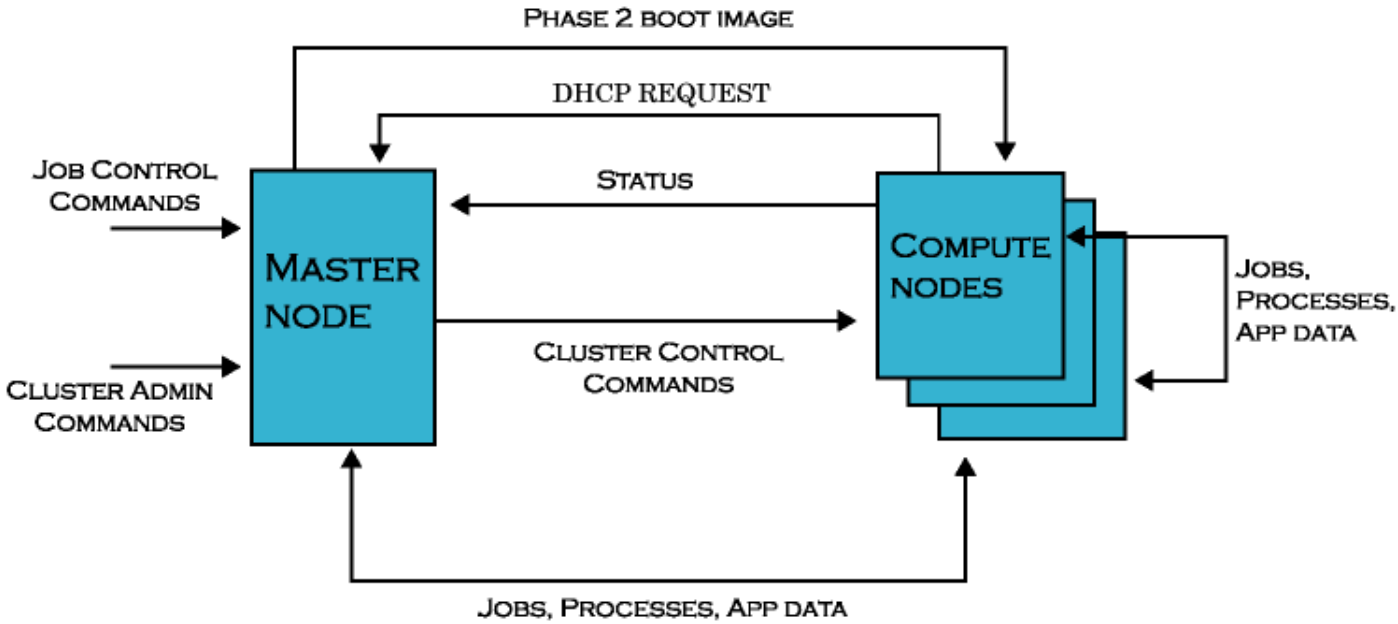


**Figure 1-5. Scyld ClusterWare Data Flow Diagram**

### Master Node to Compute Node

Following is a list of the data items sent from the master node to a compute node, as depicted in the data flow diagram.

- *Cluster control commands* — These are the commands sent from the master to the compute node telling it to perform such tasks as rebooting, halting, powering off, etc.
- *Files to be cached* — The master node send the files to be cached on the compute nodes under Scyld JIT provisioning.
- *Jobs, processes, signals, and app data* — These include the process snapshots captured by **Beowulf** for migrating processes between nodes, as well as the application data sent between jobs. **Beowulf** is the collection of software that makes up Scyld, including **beoserv** for PXE/DHCP, **BProc**, **beomap**, **beonss**, and **beostat**.
- *Final boot images* — The final boot image (formerly called the Phase 2 boot image) is sent from the master to a compute node in response to its Dynamic Host Configuration Protocol (DHCP) requests during its boot procedure.

### Compute Node to Master Node

Following is a list of the data items sent from a compute node to the master node, as depicted in the data flow diagram.

- *DHCP and PXE requests* — These requests are sent to the master from a compute node while it is booting. In response, the master replies back with the node's IP address and the final boot image.
- *Jobs, processes, signals, and app data* — These include the process snapshots captured by **Beowulf** for migrating processes between nodes, as well as the application data sent between jobs.
- *Performance metrics and node status* — All the compute nodes in a Scyld cluster send periodic status information back to the master.

### Compute Node to Compute Node

Following is a list of the data items sent between compute nodes, as depicted in the data flow diagram.

- *Jobs, processes, app data* — These include the process snapshots captured by **Beowulf** for migrating processes between nodes, as well as the application data sent between jobs.

## System Software Context

The following diagram illustrates the software components available on the nodes in a Scyld cluster.

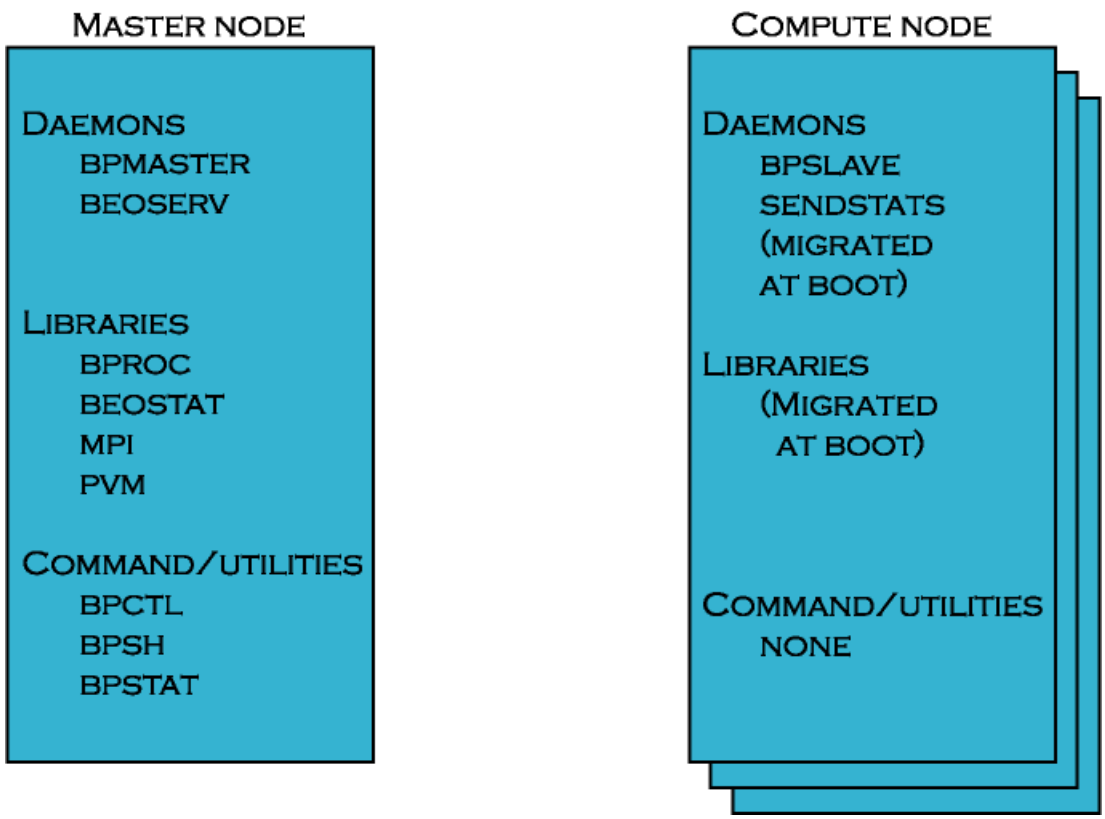


**Figure 1-6. System Software Context Diagram**

### Master Node Software Components

The master node runs the **bpmaster**, **beoserv**, and **recvstats** daemons. This node also stores the Scyld-specific libraries `libbproc` and `libbeostat`, as well as Scyld-modified versions of utilities such as MPICH, LAM, and PVM. The com-

mands and utilities are a small subset of all the software tools available on the master node.

### Compute Node Software Components

The compute nodes run the **beoclient** daemon, which serves as the init process on the compute nodes, and run the Scyld **beoklogd**, **bpslave**, and **sendstats** daemons:

- **beoklogd** is run as soon as the compute node establishes a network connection to the master node, ensuring that the master node begins capturing compute node kernel messages as early as possible.
- **bpslave** is the compute node component of **BProc**, and is necessary for supporting the unified process space and for migrating processes.
- **sendstats** is necessary for monitoring the load on the compute node and for communicating the data to the master node's **recvstats** daemon.
- **kickbackproxy** communicates with the master node's **kickbackdaemon** daemon to retrieve Name Service (NSS) information from the master node, e.g., hostnames and user names.

In general, minimal binaries reside on compute nodes, thus minimizing space consumed in a node's RAM filesystem. By default, the directories that contain common commands (e.g., `/usr/bin`) are NFS-mounted. User applications are migrated from the master node to a compute node at run-time, using a command such as **bpsh**, or are accessed using an NFS mount. Libraries are pulled to a compute node on demand, as needed.

## System Level Files

The following sections briefly describe the system level files found on the master node and compute nodes in a Scyld cluster.

### Master Node Files

The file layout on the master node is the layout of the base Linux distribution. For those who are not familiar with the file layout that is commonly used by Linux distributions, here are some things to keep in mind:

- `/bin`, `/usr/bin` — directories with user level command binaries
- `/sbin`, `/usr/sbin` — directories with administrator level command binaries
- `/lib`, `/usr/lib` — directories with static and shared libraries
- `/usr/include` — directory with include files
- `/etc` — directory with configuration files
- `/var/log` — directory with system log files
- `/var/beowulf` — directory with various ClusterWare image and status files
- `/usr/share/doc` — directory with various documentation files

Scyld ClusterWare also has some special directories and files on the master node that are useful to know about. The per-node boot logs are stored in `/var/log/beowulf/node.`*N*, where *N* is the node number. The master node's kernel and syslog messages are received by the **syslog** or **rsyslog** service, which appends these log messages to the master's `/var/log/messages` file. By default, each compute node's kernel and syslog messages are forwarded to the master node's

logging service and are also appended to the same `/var/log/messages`. However, the compute node logging can be optionally forwarded to the **syslog** or **rsyslog** service on another server. See the *syslog_server=* option in the Section called *Compute node command-line options* in Chapter 3 for details.

Configuration files for Scyld ClusterWare are found in `/etc/beowulf/`. The directory `/usr/lib/beoboot/bin/` contains various scripts that are used to configure compute nodes during boot, including the **node_up** and **setup_fs** scripts.

For more information on the special directories, files, and scripts used by Scyld ClusterWare, see Appendix A. Also see the *Reference Guide*.

### Compute Node Files

Only a very few files exist on the compute nodes. For the most part, these files are all dynamic libraries; there are almost no actual binaries. For a detailed list of exactly what files exist on the compute nodes, see Appendix A.

## Technical Description

The following sections discuss some of the technical details of a Scyld cluster, such as the compute node boot procedure, the **BProc** distributed process space and **Beowulf** process migration software, compute node categories and states, and miscellaneous components.

### Compute Node Boot Procedure

The Scyld cluster architecture is designed around light-weight provisioning of compute nodes using the master node's kernel and Linux distribution. Network booting ensures that what is provisioned to each compute node is properly version-synchronized across the cluster.

Earlier Scyld distributions supported a 2-phase boot sequence. Following PXE boot of a node, a fixed Phase 1 kernel and initial RAM disk (**initrd**) were copied to the node and installed. Alternatively, this Phase 1 kernel and **initrd** were used to boot from local hard disk or removable media. This Phase 1 boot package then built the node root filesystem **rootfs** in RAM disk, requested the run-time (Phase 2) kernel and used 2-Kernel-Monte to switch to it, then loaded the Scyld daemons and initialized the **BProc** system. Means were provided for installing the Phase 1 boot package on local hard disk and on removable floppy and CD media.

Beginning with Scyld 30-series, PXE is the supported method for booting nodes into the cluster. For some years, all servers produced have supported PXE booting. For servers that cannot support PXE booting, Scyld ClusterWare provides the means to easily produce Etherboot media on CD to use as compute node boot media. ClusterWare can also be configured to boot a compute node from a local disk. See Chapter 7 for details.

### The Boot Package

The compute node boot package consists of the kernel, **initrd**, and **rootfs** for each compute node. The **beoboot** command builds this boot package.

By default, the kernel is the one currently running on the master node. However, other kernels may be specified to the **beoboot** command and recorded on a node-by-node basis in the Beowulf configuration file. This file also includes the kernel command line parameters associated with the boot package. This allows each compute node to potentially have a unique kernel, **initrd**, **rootfs**, and kernel command lines.

> **Caution**
>
> Note that if you specify a different kernel to boot specific compute nodes, these nodes cannot be part of the **BProc** unified process space.

The path to the **initrd** and **rootfs** are passed to the compute node on the kernel command line, where it is accessible to the booting software.

Each time the ClusterWare service restarts on the master node, the **beoboot** command is executed to recreate the default compute node boot package. This ensures that the package contains the same versions of the components as are running on the master node.

### Booting a Node

A compute node begins the boot process by sending a PXE request over the cluster private network. This request is handled by the **beoserv** daemon on the master node, which provides the compute node with an IP address and (based on the contents of the Beowulf configuration file) a kernel and **initrd**. If the cluster config file does not specify a kernel and **initrd** for a particular node, then the defaults are used.

The cluster config file specifies the path to the kernel, the **initrd**, and the **rootfs**. The **initrd** contains the minimal set of programs for the compute node to establish a connection to the master and request additional files. The **rootfs** is an archive of the root filesystem, including the filesystem directory structure and certain necessary files and programs, such as the **bproc**, **filecache**, and **task_packer** kernel modules and **bpslave** daemon.

The **beoserv** daemon logs its dialog with the compute node, including its MAC address, all of the node's requests, and the responses. This facilitates debugging of compute node booting problems.

### The initrd and beoclient

Once the **initrd** is loaded, control is transferred to the kernel. Within the Scyld architecture, booting is tightly controlled by the compute node's **beoclient** daemon, which also serves as the compute node's **init** process. The **beoclient** daemon uses configuration files and executable binaries in the initrd and initial root filesystem to load the the necessary kernel modules to establish the TCP/IP connection back to the master node and basic access to local harddrives, and starts various other daemons, such as **beoklogd**, which serves as the node's local system log server to forward kernel and syslog messages (prefixed with the identify of the compute node) to the cluster's syslog server, and the **bpslave** daemon. Once **beoclient** has initialized this basic BProc functionality, then the remaining boot sequence is directed by and controlled by the master node through the **node_up** and **setup_fs** scripts and various configuration files, bootstrapping on top of the BProc functionality now executing on the node.

The **beoklogd** daemon normally forwards the kernel and syslog messages from the compute node to the master node's **syslog** or **rsyslog** service. However, this compute node logging can be optionally directed to an alternate server. See the *syslog_server=* option in the Section called *Compute node command-line options* in Chapter 3 for details. To facilitate debugging node booting problems, the kernel logging daemon on a compute node is started as soon as the network driver is loaded and the network connection to the syslog server is established.

### The rootfs

Once the network connection to the master node is established and kernel logging has been started, **beoclient** requests the **rootfs** archive, using the path passed on the kernel command line. **beoserv** provides the **rootfs** tarball, which is then uncompressed and expanded into a RAM disk.

### bpslave

The **bpslave** daemon establishes a connection to **bpmaster** on the master node, and indicates that the compute node is ready to begin accepting work. **bpmaster** then launches the **node_up** script, which runs on the master node but completes initialization of the compute node using the **BProc** commands (**bpsh**, **bpcp**, and **bpctl**).

## BProc Distributed Process Space

Scyld **Beowulf** is able to provide a single system image through its use of **BProc**, the Scyld process space management kernel enhancement. **BProc** enables the processes running on cluster compute nodes to be visible and manageable on the master node. Processes start on the master node and are migrated to the appropriate compute node by **BProc** process migration code. Process parent-child relationships and UNIX job control information are maintained with the migrated jobs, as follows:

- All processes appear in the master node's process table.
- All standard UNIX signals (kill, suspend, resume, etc.) can be sent to any process on a compute node from the master.
- The *stdin*, *stdout* and *stderr* output from jobs is redirected back to the master through a socket.

**BProc** is one of the primary features that makes a Scyld cluster different from a traditional Beowulf cluster. It is the key software component that makes compute nodes appear as attached computational resources to the master node. The figure below depicts the role **BProc** plays in a Scyld cluster.

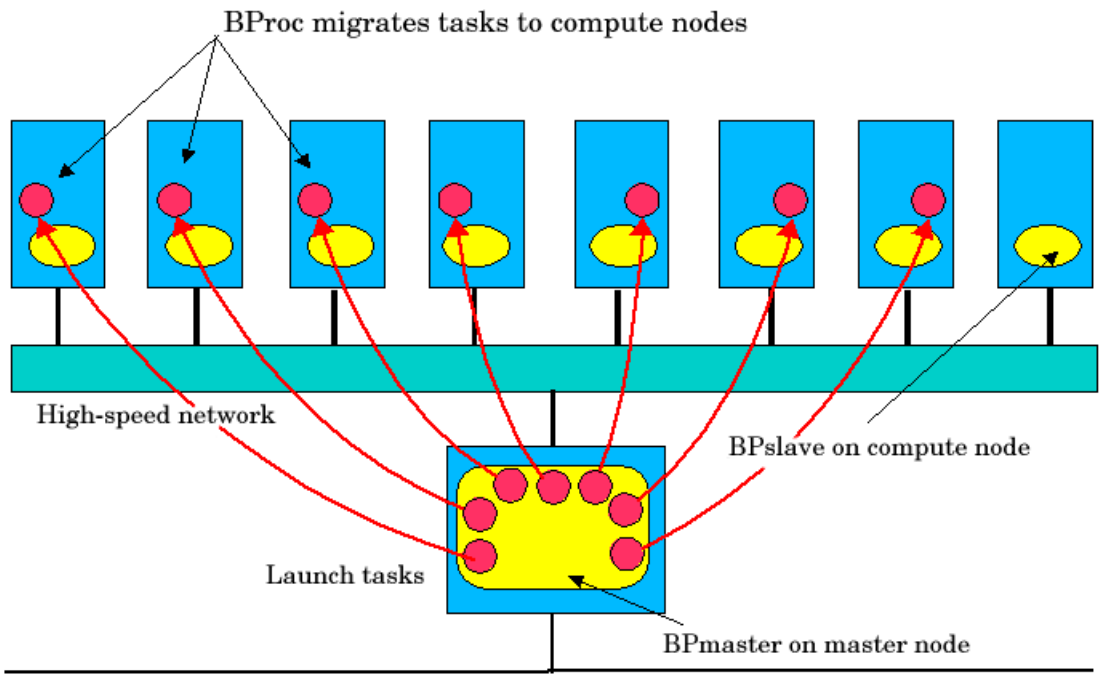


**Figure 1-7. BProc Data Flows in a Scyld Cluster**

**BProc** itself is divided into three components:

- **bpmaster** — a daemon program that runs on the master node at all times
- **bpslave** — a daemon program that runs on each of the compute nodes
- **libbproc** — a library that provides a user programming interface to BProc runtime intrinsics.

The user of a Scyld cluster will never need to directly run or interact with these daemons. However, their presence greatly simplifies the task of running parallel jobs with Scyld ClusterWare.

The **bpmaster** daemon uses a process migration module (**VMADump** in older Scyld systems or **TaskPacker** in newer Scyld systems) to freeze a running process so that it can be transferred to a remote node. The same module is also used by the **bpslave** daemon to thaw the process after it has been received. In a nutshell, the process migration module saves or restores a process's memory space to or from a stream. In the case of **BProc**, the stream is a TCP socket connected to the remote machine.

**VMADump** and **TaskPacker** implement an optimization that greatly reduces the size of the memory space required for storing a frozen process. Most programs on the system are dynamically linked; at run-time, they will use **mmap** to map copies of various libraries into their memory spaces. Since these libraries are *demand* paged, the entire library is always mapped even if most of it will never be used. These regions must be included when copying a process's memory space and included again when the process is restored. This is expensive, since the C library dwarfs most programs in size.

For example, the following is the memory space for the program **sleep**. This is taken directly from `/proc/pid/maps`.

```
08048000-08049000 r-xp 00000000 03:01 288816     /bin/sleep
08049000-0804a000 rw-p 00000000 03:01 288816     /bin/sleep
40000000-40012000 r-xp 00000000 03:01 911381     /lib/ld-2.1.2.so
40012000-40013000 rw-p 00012000 03:01 911381     /lib/ld-2.1.2.so
40017000-40102000 r-xp 00000000 03:01 911434     /lib/libc-2.1.2.so
40102000-40106000 rw-p 000ea000 03:01 911434     /lib/libc-2.1.2.so
40106000-4010a000 rw-p 00000000 00:00 0
bfffe000-c0000000 rwxp fffff000 00:00 0
```

The total size of the memory space for this trivial program is 1,089,536 bytes; all but 32K of that comes from shared libraries. **VMADump** and **TaskPacker** take advantage of this; instead of storing the data contained in each of these regions, they store a reference to the regions. When the image is restored, **mmap** will map the appropriate files to the same memory locations.

In order for this optimization to work, **VMADump** and **TaskPacker** must know which files to expect in the location where they are restored. The **bplib** utility is used to manage a list of files presumed to be present on remote systems.

## Compute Node Categories

Each compute node in the cluster is classified into one of three categories by the master node: "configured", "ignored", or "unknown". The classification of a node is dictated by whether or where it is listed in one of the following files:

- The cluster config file `/etc/beowulf/config` (includes both "configured" and "ignored nodes")
- The unknown addresses file `/var/beowulf/unknown_addresses` (includes "unknown" nodes only)

When a compute node completes its initial boot process, it begins to send out DHCP requests on all the network interface devices that it finds. When the master node receives a DHCP request from a new node, the new node will automatically

be added to the cluster as "configured" until the maximum configured node count is reached. After that, new nodes will be classified as "ignored". Nodes will be considered "unknown" only if the cluster isn't configured to auto-insert or auto-append new nodes.

The cluster administrator can change the default node classification behavior by manually editing the `/etc/beowulf/config` file (discussed in the Section called *Configuring the Cluster Manually* in Chapter 3). The classification of any specific node can also be changed manually by the cluster administrator. Also see Appendix A to learn about special directories, configuration files, and scripts.

Following are definitions of the node categories.

*Configured*

A "configured" node is one that is listed in the cluster config file `/etc/beowulf/config` using the *node* tag. These are nodes that are formally part of the cluster, and are recognized as such by the master node. When running jobs on your cluster, the "configured" nodes are the ones actually used as computational resources by the master.

*Ignored*

An "ignored" node is one that is listed in the cluster config file `/etc/beowulf/config` using the *ignore* tag. These nodes are not considered part of the cluster, and will not receive the appropriate responses from the master during their boot process. New nodes that attempt to join the cluster after it has reached its maximum configured node count will be automatically classified as "ignored".

The cluster administrator can also classify a compute node as "ignored" if for any reason you'd like the master node to simply ignore that node. For example, you may choose to temporarily reclassify a node as "ignored" while performing hardware maintenance activities when the node may be rebooting frequently.

*Unknown*

An "unknown" node is one not formally recognized by the cluster as being either "configured" or "ignored". When the master node receives a DHCP request from a node not already listed as "configured" or "ignored" in the cluster configuration file, and the cluster is not configured to auto-insert or auto-append new nodes, it classifies the node as "unknown". The node will be listed in the `/var/beowulf/unknown_addresses` file.

## Compute Node States

Cluster compute nodes may be in any of several functional states, such as *down*, *up*, or *unavailable*. Some of these states are transitional (*boot* or *shutdown*); some are informational variants of the *up* state (*unavailable* and *error*). **BProc** actually handles only 3 node operational variations:

- The node is not communicating — *down*. Variations of the *down* state may record the reason, such as *halted* (known to be halted by the master) or *reboot* (the master shut down the node with a reboot command).
- The node is communicating — *up*, *[up]*, *alive*, *unavailable*, or *error*. Here the strings indicate different levels of usability.
- The node is transitioning — *boot*. This state has varying levels of communication, operating on scripted sequence.

During a normal power-on sequence, the user will see the node state change from *down* to *boot* to *up*. Depending on the machine speed, the *boot* phase may be very short and may not be visible due to the update rate of the cluster monitoring tools. All state information is reset to *down* whenever the **bpmaster** daemon is started/restarted.

In the following diagram, note that these states can also be reached via imperative commands such as **bpctl**. This command can be used to put the node into the *error* state, such as in response to an error condition detected by a script.

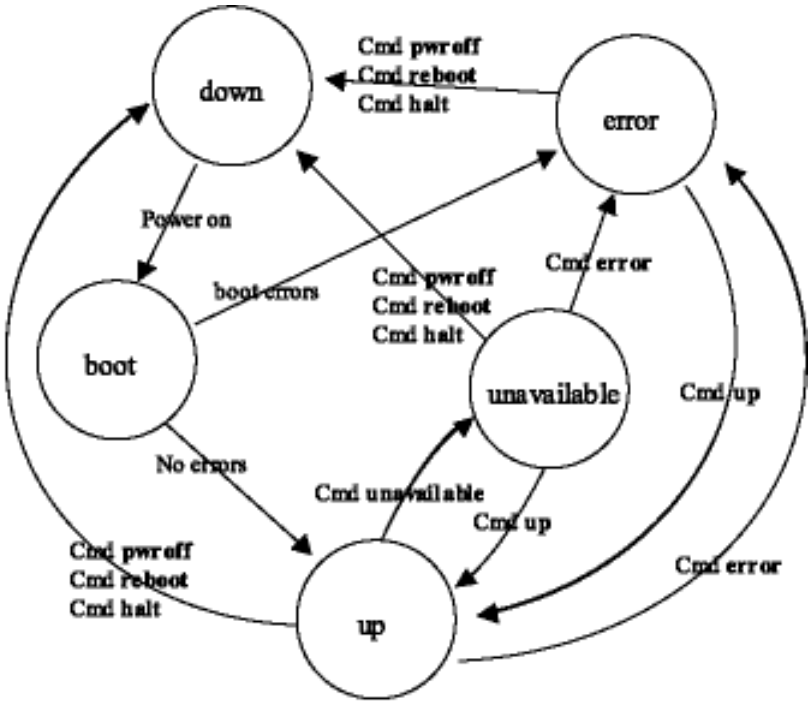


**Figure 1-8. Node State Transition Diagram**

Following are definitions of the compute node states:

**down**

From the master node's view, *down* means only that there is no communication with the compute node. A node is *down* when it is powered off, has been halted, has been rebooted, has a network link problem, or has some other hardware problem that prevents communication.

**boot**

This is a transitional state, during which the node will not accept user commands. The *boot* state is set when the **node_up** script has started and will transition to *up* or *error* when the script has completed. While in the *boot* state, the node will respond to administrator commands, but indicates that the node is still being configured for normal operation. The duration of this state varies with the complexity of the **node_up** script.

**up**

This is a functional state, set when the **node_up** script has completed without encountering any errors. **BProc** checks the return status of the script and sets the node state to *up* if the script was successful. This is the only state where the node is available to non-administrative users, as **BProc** checks this before moving any program to a node; administrator programs bypass this check. This state may also be commanded when the previous state was *unavailable* or *error*.

**error**

This is an informational state, set when the **node_up** script has exited with errors. The administrator may access the node, or look in the `/var/log/beowulf/node.`*x* (where *x* is a node number) file to determine the problem. If a problem is seen to be non-critical, the administrator may then set the node to *up*.

**unavailable**

This is a functional state. The node is not available for non-administrative users; however, it is completely available to the administrator. Currently running jobs will not be affected by a transition to this state. With respect to job control, this state comes into play only when attempting to run new jobs, as new jobs will fail to migrate to a node marked *unavailable*. This state is intended to allow node maintenance without having to bring the node offline.

**[up]**

This Scyld ClusterWare node is *up* and is being actively managed by another master node, which for now is the node's primary master. The secondary master node(s) see the node as *[up]*. A secondary master can ssh to the node (if ssh is enabled), but the node only responds to **BProc** commands from its primary master (e.g., **bpsh** and **bpcp**). See Chapter 8 for details.

**alive**

This non-Scyld node is alive to the extent that it is running the **sendstats** daemon to report various `/proc` statistics about the node state, and it is integrated as a compute node in the cluster. For example, the Job Manager may be able to run jobs on this node. See Chapter 6 and Chapter 7 for details.

## Miscellaneous Components

Scyld ClusterWare includes several miscellaneous components, such as name lookup functionality (**beonss**), IP communications ports, library caching, and external data access.

### beonss

**beonss** provides name service lookup functionality for Scyld ClusterWare. The information it provides includes hostnames, netgroups, and user information. In general, whatever name service information is available to the master node, using whatever query methods available to the master node (e.g., NIS, LDAP), is also transparently available to the compute nodes through the **beonss** functionality. The Scyld ClusterWare installation automatically (and silently) configures **beonss**.

#### Hostnames

**beonss** provides dynamically generated hostnames for all the nodes in the cluster. The hostnames are of the form *.<nodenumber>*, so the hostname for node 0 would be *.0*, the hostname for node 50 would be *.50*, and the hostname for the master node would be *.-1*.

The *nodename* entries in the `/etc/beowulf/config` file allow for the declaration of additional hostname aliases for compute nodes. For instance,

```
nodename n%N
```

declares aliases for nodes, e.g., *n4* is an alias for node *.4*. For another example, suppose the IP address of node 4 is 10.0.0.4, and suppose that node 4 has its IPMI interface configured to respond to the IP address 10.1.0.4. Then the line:

```
nodename n%N-ipmi 0.1.0.0 ipmi
```

declares aliases for the hostnames in the group called *ipmi*. The hostname *n4-ipmi* is the arithmetic sum of *n4*'s IP address 10.1.0.4 plus the offset 0.1.0.0, forming the IP address 10.1.0.4. See **man beowulf-config** and the comments in the file `/etc/beowulf/config` for details and other examples.

**beonss** also provides the hostname *master*, which always points to the IP of the master node on the cluster's internal network. The hostnames *.-1* and *master* always point to the same IP.

These hostnames will always point to the right IP address based on the configuration of your IP range. You don't need to do anything special for these hostnames to work. Also, these hostnames will work on the master node or any of the compute nodes.

Note that **beonss** does not know the hostname and IP address that the master node uses for the outside network. Suppose your master node has the public name *mycluster* and uses the IP address 1.2.3.4 for the outside network. By default, a compute node on the private network will be unable to open a connection to *mycluster* or to 1.2.3.4. However, by enabling IP forwarding in both the `/etc/beowulf/config` file and the `/etc/sysctl.conf` file, compute nodes can resolve hostnames and access hosts that are accessible by the master through the master's public network interface, provided you have your DNS services working and available on the compute nodes.

**Tip:** When you enable IP forwarding, the master node will set up NAT routing between your compute nodes and the outside world, so your compute nodes will be able to make outbound connections. However, this does not enable outsiders to access or "see" your compute nodes.

**Caution**

On compute nodes the NFS directories must be mounted using either the NFS server's IP address or the "$MASTER" keyword, as is specified in the `/etc/beowulf/fstab` file. Hostnames cannot be used because the compute node's NFS mounting is performed before the node's name service is active, which would otherwise be able to translate a hostname to its IP address.

#### Netgroups

Netgroups are a concept from NIS. They make it easy to specify an arbitrary list of machines, then treat all those machines the same when carrying out an administrative procedure (for example, specifying what machines to export NFS filesystems to).

**beonss** creates one netgroup called *cluster*, which includes all of the nodes in the cluster. This is used in the default `/etc/exports` file in order to easily export `/home` to all of the compute nodes.

#### User Information

When jobs are running on the compute nodes, **beonss** allows the standard **getpwnam()** and **getpwuid()** functions to successfully retrieve information (such as username, home directory, shell, and uid), as long as these functions are retrieving information on the user that is running the program. All other information that **getpwnam()** and **getpwuid()** would normally retrieve will be set to "NULL".

### IP Communications Ports

Scyld ClusterWare uses a few TCP/IP and UDP/IP communication ports when sending information between nodes. Normally, this should be completely transparent to the user. However, if the cluster is using a switch that blocks various ports, it may be important to know which ports are being used and for what.

Following are key components of Scyld ClusterWare and the ports they use:

- **beoserv** — This daemon is responsible for replying to the DHCP request from a compute node when it is booting. The reply includes a new kernel, the kernel command line options, and a small final boot RAM disk. The daemon supports both multi-cast and uni-cast file serving.

By default, **beoserv** uses TCP port 932. This can be overridden by changing the value of the *server beofs2* directive (formerly *server tcp*, which is deprecated but continues to be accepted) in the `/etc/beowulf/config` file to the desired port number.

- **BProc** — This ClusterWare component provides unified process space, process migration, and remote execution of commands on compute nodes. By default, **BProc** uses TCP port 933. This can be overridden by changing the value of the *server bproc* directive in the `/etc/beowulf/config` file to the desired port number.
- **BeoStat** — This service is composed of compute node daemons (**sendstats**), a master node daemon (**recvstats**), and a master node library (**libbeostat**) that collects performance metrics and status information from compute nodes and transmits this information to the master node for cacheing and for distribution to the various cluster monitoring display tools. The daemons use UDP port 5545 by default.

## Library Caching

One of the features Scyld ClusterWare uses to improve the performance of transferring jobs to and from compute nodes is to cache libraries. When **BProc** needs to migrate a job between nodes, it uses the process migration code (**VMADump** or **TaskPacker**) to take a snapshot of all the memory the process is using, including the binary and shared libraries. This memory snapshot is then sent across the private cluster network during process migration.

**VMADump** and **TaskPacker** take advantage of the fact that libraries are being cached on the compute nodes. The shared library data is *not* included in the snapshot, which reduces the amount of information that needs to be sent during process migration. By not sending over the libraries with each process, Scyld ClusterWare is able to reduce network traffic, thus speeding up cluster operations.

## External Data Access

There are several common ways for processes running on a compute node to access data stored externally to the cluster, as discussed below.

### Transfer the data

You can transfer the data to the master node using a protocol such as **scp** or **ftp**, then treat it as any other file that resides on the master node.

### Access the data through a network filesystem, such as NFS or AFS

Any remote filesystem mounted on the master node can't be re-exported to the compute node. Therefore, you need to use another method to access the data on the compute nodes. There are two options:

- Use **bpsh** to start your job, and use shell redirection on the master node to send the data as **stdin** for the job
- Use MPI and have the rank 0 job read the data, then use MPI's message passing capabilities to send the data.

If you have a job that is natively using **Beowulf** functions, you can also have your job read the data on the master node before it moves itself to the compute nodes.

#### NFS mount directories from external file servers

There are two options:

- For file servers directly connected to the cluster private network, this can be done directly, using the file server's IP address. Note that the server name cannot be used, because the name service is not yet up when `/etc/beowulf/fstab` is evaluated.
- For file servers external to the cluster, setting up IP forwarding on the master node allows the compute nodes to mount exported directories using the file server's IP address.

#### Use a cluster filesystem

If you have questions regarding the use of any particular cluster filesystem with Scyld ClusterWare, contact Scyld Customer Support for assistance.

## Software Components

The following sections describe the various software packages in Scyld ClusterWare, along with their individual components. For additional information, see the *Reference Guide*.

### BeoBoot Tools

The following tools are associated with the **beoboot** package. For additional information, see the *Reference Guide*.

**BeoBoot**

This utility is used to generate boot images for the compute nodes in the cluster. Earlier versions of Scyld used two types of images, initial (Phase 1) and final (Phase 2). The initial images were placed on the hard disk or a floppy disk, and were used to boot the nodes. The final image was downloaded from the master node by the initial image. Currently, only the final image is used by Scyld ClusterWare; support for initial images has been dropped.

By default, the final image is stored on the master node in the `/var/beowulf/boot.img` file; this is where the **beoserv** daemon expects to find it. Where initial images were used to begin the network boot process for systems that lacked PXE support, Scyld now provides PXELinux for this purpose. Bootable PXELinux media may be created for CD-ROM booting.

**beoserv**

This is the **BeoBoot** daemon. It responds to DHCP requests from the compute nodes in the cluster and serves them their final boot images over the private cluster network.

### BProc Daemons

The following daemons are associated with **BProc**. For additional information, see the *Reference Guide*.

**bpmaster**

This is the **BProc** master daemon. It runs on the master node, listening on a TCP port and accepting connections from **bpslave** daemons. Configuration information comes from the `/etc/beowulf/config` file.

**bpslave**

This is the **BProc** compute daemon. It runs on a compute node to accept jobs from the master, and connects to the master through a TCP port.

## BProc Clients

The following command line utilities are closely related to **BProc**. For additional information, see the *Reference Guide*.

**bpsh**

This is a replacement for **rsh** (remote shell). It runs a specified command on an individually referenced node. The "nodenum" parameter may be a single node number, a comma delimited list of nodes, "-a" for all nodes that are up, or "-A" for all nodes that are not down.

**bpsh** will forward standard input, standard output, and standard error for the remote processes it spawns. Standard output and error are forwarded subject to specified options; standard input will be forwarded to the remote processes. If there is more than one remote process, standard input will be duplicated for every remote node. For a single remote process, the exit status of **bpsh** will be the exit status of the remote process.

**bpctl**

This is the **BProc** control utility. It can be used to apply various commands to individually referenced nodes. **bpctl** can be used to change the user and group ownership settings for a node; it can also be used to set a node's state. Finally, this utility can be used to query such information as the node's IP address.

**bpcp**

This utility can be used to copy files between machines in the cluster. Each file (f1...fn) or directory argument (dir) is either a remote file name of the form *node:path*, or a local file name (containing no colon ":" characters).

**bpstat**

This command displays various pieces of status information about the compute nodes. The display is formatted in columns specifying node number, node status, node permission, user access, and group access. This program also includes a number of options intended to be useful for scripts.

## ClusterWare Utilities

Following are various command line and graphical user interface (GUI) utilities that are part of Scyld ClusterWare. For additional information, see the *Reference Guide*.

**beostat**

The **beostat** command line tool is a text-based utility used to monitor cluster status and performance. This tool provides a text listing of the information from the `/proc` structure on each node. See the Section called *beostat* in Chapter 2 for a discussion of this tool.

**beostatus**

The **beostatus** GUI tool is used to monitor cluster status and performance. See the Section called *beostatus* in Chapter 2 for a discussion of this tool.

# Chapter 2. Monitoring the Status of the Cluster

Scyld ClusterWare provides several methods to monitor cluster performance and health, with a Web browser, a GUI, the command line, and "C" language interfaces. In general, these tools provide easy access to the information available through the Linux **/proc** filesystem, as well as **BProc** information for each of the cluster nodes. The monitoring programs are available to both administrators and regular users, since they provide no cluster command capabilities.

## Monitoring Utilities

### Cluster Monitoring Interfaces

Scyld ClusterWare provides several cluster monitoring interfaces. Following is brief summary of these interfaces; more detailed information is provided in the sections that follow:

- **libbeostat** — The **libbeostat** library, together with the compute nodes' **sendstats** daemons and the master node's **recvstats** daemon, provides the underpinning for the various display tools. Users can also create custom displays or create more sophisticated resource scheduling software by interfacing directly to libbeostat.
- **beostat** — The **beostat** command provides a detailed command-line display using the underlying libbeostat library. With no options, **beostat** lists information for the master node and all compute nodes that is retrieved from `/proc/cpuinfo`, `/proc/meminfo`, `/proc/loadavg`, `/proc/net/dev`, and `/proc/stat`. Alternatively, you can use the arguments to select any combination of those statistics.
- **beostatus** — The **beostatus** cluster monitoring utility uses the underlying libbeostat functionality to display CPU utilization, memory usage, swap usage, disk usage, and network utilization. It defaults to a bar graph X-window GUI, but can display the information in several text formats. For large clusters, a small footprint GUI can be selected, with colored dots depicting the overall status on each node.
- **bpstat** — This displays a text-only snapshot of the current cluster state. The bpstat utility only reports nodes that are part of the BProc unified process space, vs. beostat and beostatus, which report on all nodes (BProc and non-BProc) that execute a sendstats daemon.
- **Ganglia** — Scyld installs the popular **Ganglia** monitoring package by default, but does not configure it to execute by default. For information on configuring Ganglia, see the Section called *Ganglia*.
- **beoweb** — **Beoweb** is an optional Web service that can execute on the cluster's master node. Built with **Pylons** (a Python-based Web framework), beoweb exposes an API for cluster status and remote job submission and monitoring.
- **SNMP** — Scyld ClusterWare HPC includes an enhanced distribution of the Open Source **Simple Network Management Protocol (SNMP)** suite of applications.

### Monitoring Daemons

Underlying the libbeostat monitoring facility are two daemons: **sendstats** and **recvstats**. The **recvstats** daemon is started by the `/etc/rc.d/init.d/clusterware` script and only executes on the master node. A **sendstats** daemon executes on each compute node and sends status information at regular intervals (currently once per second) to the master's **recvstats** daemon. For more information on the daemon options, see **man recvstats** and **man sendstats**, or the *Reference Guide*.

The optional **beoweb** service employs the **paster** daemon on the master node. See the Section called *beoweb* for details.

The optional **snmpd-scyld** service employs the **snmpd-scyld** daemon, and the optional **snmptrapd-scyld** service employs the **snmptrapd-scyld** daemon, both of which execute on the master node. See the Section called *SNMP* for details.

## Using the Data

The outputs from the monitoring utilities can provide insights into obtaining the best performance from your cluster. If you are new to cluster computing, you will want to note the relationship between the different machine resources, including CPU utilization, swap usage, and network utilization. Following are some useful guidelines:

- Low CPU usage with high network traffic might indicate that your system is I/O bound and could benefit from faster network components.
- Low network load and high CPU usage indicate that your system performance could improve with faster CPUs.
- Medium to high swap usage is always bad. This indicates that memory is oversubscribed, and application pieces must be moved to the much slower disk sub-system. This can be a substantial bottleneck, and is a sure sign that additional RAM is needed.

Any of these issues could be helped with application optimization, but sometimes it is more economical to add resources than to change working software.

For best performance of a computational workload, make sure your compute nodes have ample memory for the application and problem set. Also, use diskless compute nodes or configure local disks for scratch file space rather than swap space.

## beostatus

The **beostatus** GUI display is a Gnome X-window that supports four different types of display generation, all of which can be operated simultaneously. Output in bar graph mode (also known as "Classic" mode) is the default, and is provided by a Gnome/GTK+ GUI display. This display is updated once every 5 seconds by default, but the update rate may be changed using the **-u** option.

You can start **beostatus** by clicking the "blocks" icon on the desktop.

Alternatively, type the command **beostatus** in a terminal window on the master node; you do not need to be a privileged user to use this command.

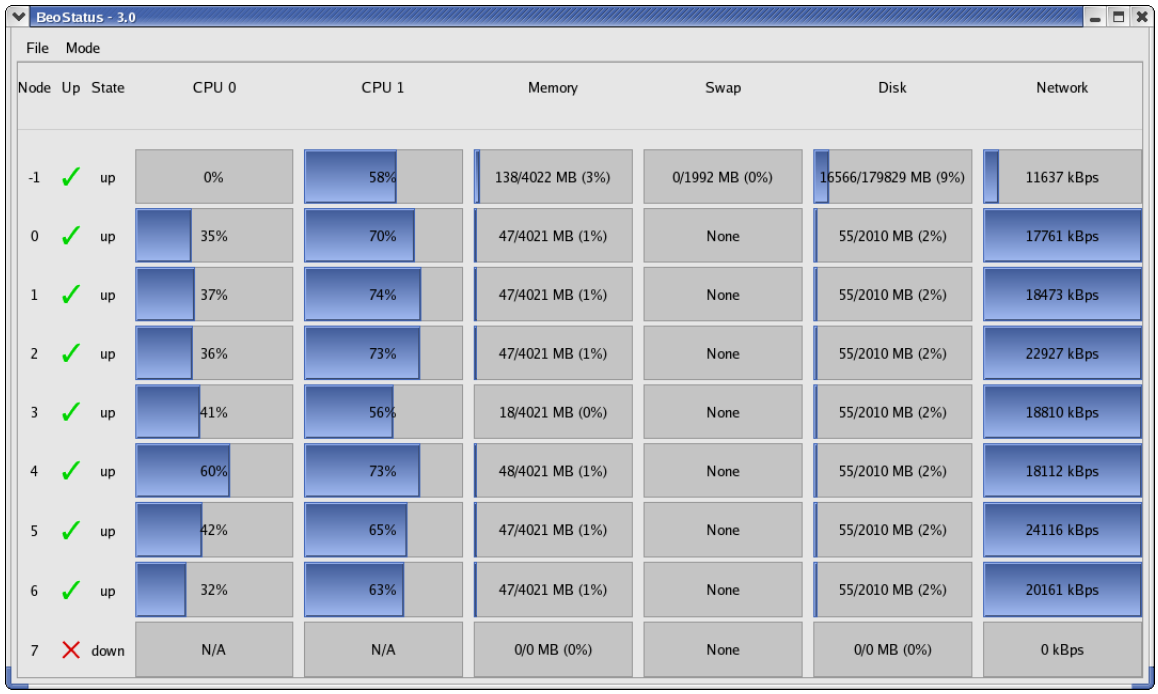


**Figure 2-1. beostatus GUI Display (in "Classic" Mode)**

## beostatus File Menu

The **File** menu in the **beostatus** GUI display includes two options, **Preferences** and **Quit**, as described below.

### Preferences

Selecting **Preferences** from the **File** menu displays the **Options** dialog box (shown below). You can change the values for update rate, master node maximum bandwidth, slave (compute node) maximum bandwidth, and the default display mode.

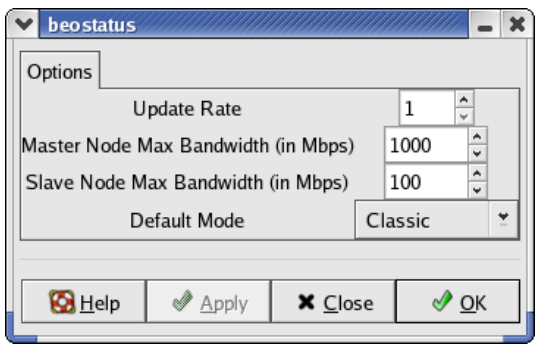


**Figure 2-2. beostatus Preference Options**

### Quit

Selecting **Quit** from the **File** menu closes the **beostatus** GUI display.

## beostatus Modes

The **Mode** menu in the **beostatus** GUI display allows you to choose between the various display options.

Some display options can also be accessed using command line options, including *Dots* mode, *Curses* mode, and *Text* mode. These display options are described in the following sections.

### beostatus *Dots* Mode

Output in *Dots* mode (**beostatus -d**) provides a Gnome/GTK+ GUI display. Each node is represented by a colored dot. This output provides a small "footprint", intended for quick overviews and for situations where the screen size needed for the full display for large clusters is unavailable.

**Figure 2-3. beostatus GUI Display in *Dots* Mode**

Following are the color indicators used in *Dots* mode:

- **Red** — No access; node state *down*
- **Yellow** — Admin access only; node state *unavailable*, *boot*, or *error*
- **Green** — Ready; node state *up* and node load less than/equal 48%
- **Blue** — Busy; node state *up* and node load greater than 48%

Note that SMP is considered for node load calculation as load(CPU1) + load(CPU2) > 48%.

### beostatus *Curses* Mode

Output in *Curses* mode (**beostatus -c**) prints a column header and a line for each node without a linefeed. This continuous output provides a method to monitor the system over text-only connections, such as the installed **ssh** server. Following is an example of the output in *Curses* mode:

```
BeoStatus - 3.0
Node State CPU 0 Memory Swap Disk Network
 -1 up 2.5% 91.7% 0.0% 9.2% 1 kBps
  0 up 0.2% 20.5% 0.0% 25.0% 1 kBps
  1 up 0.1% 20.5% 0.0% 25.0% 1 kBps
  2 up 0.1% 20.5% 0.0% 25.0% 1 kBps
  3 up 0.2% 20.4% 0.0% 25.0% 1 kBps
```

```
  4 up 0.1% 20.3% 0.0% 25.0% 1 kBps
  5 up 0.1% 20.3% 0.0% 25.0% 1 kBps
  6 up 0.2% 20.6% 0.0% 25.0% 1 kBps
  7 up 0.1% 20.4% 0.0% 25.0% 1 kBps
```

## beostat

The **beostat** utility is a command-line program that provides a text listing of the information from **/proc** on each node. Following is example output from a single node.

```
============== Node: .0 (index 0) ==================

 *** /proc/cpuinfo *** Tue Sep 12 14:38:04 2006
num processors  : 2
vendor_id       : AuthenticAMD
cpu family      : 15
model           : 5
model name      : AMD Opteron(tm) Processor 248
stepping        : 10
cpu MHz         : 2211.355
cache size      : 1024 KB
fdiv_bug        : no
hlt_bug         : no
sep_bug         : no
f00f_bug        : no
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     : 1
wp              : yes
bogomips        : 4422.05

 *** /proc/meminfo *** Tue Sep 12 14:38:04 2006
        total:    used:    free:  shared: buffers:  cached:
Mem:  4216758272 18960384 4197797888        0        0        0
Swap:        0        0        0
MemTotal:   4117928 kB
MemFree:    4099412 kB
MemShared:        0 kB
Buffers:          0 kB
Cached:           0 kB
SwapTotal:        0 kB
SwapFree:         0 kB

 *** /proc/loadavg *** Tue Sep 12 14:38:04 2006
0.00 0.00 0.00 28/28 0

 *** /proc/net/dev *** Tue Sep 12 14:38:04 2006
Inter-|   Receive                                                |  Transmit
 face |bytes packets errs drop fifo frame compressed multicast | bytes packets errs drop fifo colls
  eth0:105630479    73832        0        0        0     0          0         0  5618030  35864
  eth1:       0        0        0        0        0     0          0         0        0     0
```

```
 *** /proc/stat ***
cpu0 0 0 115 203357             Tue Sep 12 14:38:04 2006
cpu1 4 0 125 203261             Tue Sep 12 14:38:04 2006

 *** statfs ("/") *** Tue Sep 12 14:38:04 2006
path:           /
f_type:         0x1021994
f_bsize:        4096
f_blocks:       514741
f_bfree:        500311
f_bavail:       500311
f_files:        514741
f_ffree:        514630
f_fsid:         000000 000000
f_namelen:      255
```

The **libbeostat** library contains the "C" language functions listed below. You compile with the header files `sys/bproc.h` and `sys/beostat.h`, adding the linker commands **-lbproc -lbeostat**.

```
beostat-get-cpu-count
beostat-get-name
beostat-get-time
beostat-get-cpuinfo-x86
beostat-get-meminfo
beostat-get-loadavg
beostat-get-net-dev
beostat-get-stat-cpu
beostat-get-MHz
beostat-get-statfs-p
beostat-get-last-multicast
beostat-set-last-multicast
beostat-get-cpu-percent
beostat-get-net-rate
beostat-get-disk-usage
beostat-count-idle-cpus
beostat-count-idle-cpus-on-node
beostat-get-avail-nodes-by-id
beostat-is-node-available
```

## bpstat

**bpstat** displays a text-only snapshot of the current cluster state/configuration:

```
[root@cluster ~] # bpstat
Node(s) Status Mode  User Group
16-31 down ---------- root root
0-15 up ---x--x--x root root
```

You can include the master node in the display, which is especially useful if the master node has non-default access permissions:

```
[root@cluster ~] # bpstat
Node(s) Status Mode  User Group
```

```
16-31 down ---------- root root
-1 up ---x--x--- root root
0-15 up ---x--x--x root root
```

Using the **-p** option, you can view the PID for each user process running on the nodes. You can then pipe the **ps** command into **grep** to get the command string associated with it, such as `ps -aux |grep 8370`. Normal process signaling will work with these PIDs, such as `kill -9 8369`.

```
PID Node Ghost
8367 0 -1
8368 1 -1
8369 2 -1
8370 3 -1
```

See the *Reference Guide* for more details on the command options.

## Ganglia

**Ganglia** is an open source distributed monitoring technology for high-performance computing systems, such as clusters and grids. In current versions of Scyld ClusterWare, **Ganglia** provides network metrics for the master node, time and string metrics (`boottime`, `machine_type`, `os_release`, and `sys_clock`), and constant metrics (`cpu_num` and `mem_total`). **Ganglia** uses a web server to display these statistics; thus, to use **Ganglia**, you must run a web server on the cluster's master node.

When installing Scyld ClusterWare, make sure the **Ganglia** package is selected among the package groups to be installed. Once you have completed the Scyld installation and configured your compute nodes, you will need to configure **Ganglia** as follows:

1. Name your cluster.

   By default, **Ganglia** will name your cluster "my cluster". You should change this to match the master node's hostname. In the file `/etc/ganglia/gmetad.conf`, and on or about line 44, change:

   ```
   data_source "my cluster" localhost
   ```
   to replace *my cluster* with the master's hostname. Note that **Ganglia** will not collect or display statistics without at least one entry for `data_source`.

2. Enable and start the **Ganglia Data Collection Service**.

   ```
   [root@cluster ~] # chkconfig beostat on
   [root@cluster ~] # systemctl enable  xinetd
   [root@cluster ~] # systemctl enable  httpd
   [root@cluster ~] # systemctl enable  gmetad
   [root@cluster ~] # systemctl restart xinetd
   [root@cluster ~] # systemctl start   httpd
   [root@cluster ~] # systemctl start   gmetad
   ```

3. Visit `http://localhost/ganglia` in a web browser.

   Note that if you are visiting the web page from a computer other than the cluster's master node, then you must change `localhost` to the master node's hostname. For example, if the hostname is "iceberg", then you may need to use its fully qualified name, such as `http://iceberg.penguincomputing.com/ganglia`

> **Caution**
>
> The **Ganglia** graphs that track load (1-, 5-, and 15-minute), the number of CPUs, and the number of processes may appear inaccurate. These graphs are in fact reporting correct statistics, but for the system as a whole rather than just user processes. Scyld draws its statistics directly from system data structures and /**proc**. It does not take any further steps to interpret or post-process the metrics reported by these data structures.

## beoweb

The **beoweb** service does not execute by default. To enable it:

```
chkconfig beoweb on
```

and then it will start automatically the next time the master node boots. It can be started immediately by doing:

```
[root@cluster ~] # service beoweb start
```

Beoweb exposes an API for cluster status monitoring and remote job submission and monitoring. In its current state, beoweb is best used when paired with **PODTools** to enable remote job submission. (See the User's Guide for details about PODTools.) Beoweb does not yet support being viewed with a web browser; rather, it merely provides a web service accessible through APIs. Beoweb supports job submission using the TORQUE, Slurm, or SGE resource manager.

Beoweb is installed in `/opt/scyld/beoweb`, and the main configuration file, `beoweb.ini`, is located there. Some key settings to inspect are:

- **host = 0.0.0.0**

  This specifies the interface on which Beoweb will bind/listen. *0.0.0.0* specifies all available interfaces. Use an actual IP address to limit this to a single interface.

- **port = 5000**

  The port number on which beoweb listens. Change to a different port number as needed.

- **ssl_pem = %(here)s/data/beoweb.pem**

  The ssl_pem parameter controls whether or not beoweb uses SSL/TLS encryption for communication. It is strongly encouraged that you use SSL. When beoweb is installed, a temporary PEM file will be created at %(here)s/data/beoweb.pem. This certificate is good for 365 days.

- **auth.use_system_shadow = True**

  The value defaults to True. Unless explicitly disabled, beoweb will read `/etc/shadow` for user authentication. If this is set to False, you must use **auth.auth_file** to specify a different list of authorized users.

- **auth.auth_file = %(here)s/data/shadow**

  This file allows for user passwords to be stored independently from the master node's `/etc/shadow` file. Currently, beoweb only supports shadow-type login accounts. For example, if you put user credentials in `%(here)s/data/shadow` and not in `/etc/shadow`, then that user can access the master node's beoweb services without being allowed to actually login to the master node. The format for this file is identical to `/etc/shadow`.

- **stage.jobs_dir = podsh_jobs**

  This names a folder that will be created and used in a user's home directory for job scripts uploaded through PODTools.

- **stage.port_range = 10000-11000**

  When file uploads and downloads are requested through beoweb using PODTools, the files are transfered through a TCP socket connection. Beoweb opens a socket on the port in the range given in this entry, then sends that port number back to PODTools for use. This range should be chosen such that it does not conflict with other services on your system.

## SNMP

Scyld ClusterWare HPC includes **net-snmp-scyld**, which is a redistribution of the Open Source **Simple Network Management Protocol (SNMP)** suite of applications that contains a Scyld MIB implementation. It installs into `/opt/scyld/snmp/` and can co-exist with the base distribution's **net-snmp**, **net-snmp-devel**, **net-snmp-libs**, and **net-snmp-utils** packages. Visit http://www.net-snmp.org/[1] for general information about how to configure and use SNMP.

To manually execute the Scyld versions of the commands, use an environment module to set up the various search paths that will discover the Scyld files before discovering the base distribution files:

```
[root@cluster ~] # module load net-snmp-scyld
```

To enable the **snmpd-scyld** service, first set up the root SNMP configuration. An example of the configuration file is:

```
[root@cluster ~] # cat /root/.snmp/snmpd.conf
rocommunity  public     localhost
rwcommunity  demopublic localhost

trapsink     localhost  public
trap2sink    localhost  secret
informsink   localhost

# Uncomment to set SCYLD-MIB notification objects default values
# when snmpd starts

#cwtrapenable 2      # 1 is enable, 2 is disable
#cwminfreemem 20
#cwmaxdiskusage 90
```

To enable the **snmptrapd-scyld** service, first set up the root SNMP trap configuration. An example of the configuration file is:

```
[root@cluster ~] # cat root/.snmp/snmptrapd.conf
authCommunity log,execute,net public
```

Finally, enable the Scyld SNMP services, as desired. In generally, you will first want to disable the equivalent base distribution services, although this may not be necessary:

```
[root@cluster ~] # service snmpd stop

[root@cluster ~] # service snmptrapd stop

[root@cluster ~] # chkconfig snmpd off

[root@cluster ~] # chkconfig snmptrapd off
```

Then enable the Scyld services:

```
[root@cluster ~] # chkconfig snmpd-scyld on
```

```
[root@cluster ~] # chkconfig snmptrapd-scyld on

[root@cluster ~] # service snmpd-scyld start

[root@cluster ~] # service snmptrapd-scyld start
```

## Notes

1. http://www.net-snmp.org

# Chapter 3. Configuring the Cluster

The Scyld ClusterWare configuration is defined by the contents of several flat ASCII files. Most of these files reside in the `/etc/beowulf/` directory. Various ClusterWare scripts (which mostly reside in `/usr/lib/beoboot/bin`), daemons, and commands read (and some occasionally update) these flat files.

The *root* can manipulate the configuration manually using your favorite editor.

## Configuring the Cluster Manually

This section discusses how to configure a cluster. Manual editing of configuration files, especially the centerpiece `/etc/beowulf/config` file, should only be done with care, together with sufficient understanding of the ramifications of the manual manipulations.

> **Caution**
>
> If manual edits are made to the `config` file for a running cluster, then as soon as you have finished editing and saving the file, be sure to execute the command: `service clusterware reload` which will immediately notify the **beoserv** and **bpmaster** daemons to re-read the file by sending a *SIGHUP* signal to each.

### Configuration Files

#### /etc/beowulf/config

The file `/etc/beowulf/config` is the principal configuration file for the cluster. The `config` file is organized using keywords and values, which are used to control most aspects of running the cluster, including the following:

- The name, IP address and netmask of the network interface connected to the private cluster network
- The network port numbers used by ClusterWare for various services
- The IP address range to assign to the compute nodes
- The MAC (hardware) address of each identified node accepted into the cluster
- The node number and IP address assigned to each hardware address
- The default kernel and kernel command line to use when creating a boot file
- A list of kernel modules to be available for loading on compute nodes at runtime
- A list of shared library directories to cache on the compute nodes
- A list of files to prestage on the compute nodes
- Compute node filesystem startup policy
- The name of the final boot file to send to the compute nodes at boot time
- The hostname and hostname aliases of compute nodes
- Compute node policies for handling local disks and filesystems, responding to master node failure, etc.

The following sections briefly discuss some key aspects of the configuration file. See the *Reference Guide* (or **man beowulf-config**) for details on the specific keywords and values in `/etc/beowulf/config`.

### Setting the IP Address Range

The IP address range should be kept to a minimum, as all the cluster utilities will loop through this range. Having a few spare addresses is a good idea to allow for growth in the cluster. However, having a large number of addresses that will never be used will be an unnecessary waste of resources.

### Identifying New Nodes

When a new node boots, it issues a DHCP request to the network in order to get an IP address assigned to it. The master's **beoserv** detects these DHCP packets, and its response is dependent upon the current *nodeassign* policy. With a default *append* policy, **beoserv** appends a new *node* entry to the end of the `/etc/beowulf/config` file. This new entry identifies the node's MAC address(es), and the relative ordering of the *node* entry defines the node's number and what IP address is assigned to it. With a *manual* policy, **beoserv** appends the new node's MAC address to the file `/var/beowulf/unknown_addresses`, and then assigns a temporary IP address to the node that is outside the *iprange* address range and which does not integrate this new node into the cluster. It is expected that the cluster administrator will eventually assign this new MAC address to a cluster node, giving it a *node* entry with an appropriate position and node number. Upon cluster restart, when the node reboots (after a manual reset or an IPMI powercycle), the node will assume its assigned place in the cluster. With a *locked* policy, the new node gets ignored completely: no recording of its MAC address, and no IP address assignment.

### Assigning Node Numbers and IP Addresses

Two `config` file keywords control the assignment of IP addresses to compute nodes on the private cluster network: *nodes* and *iprange*. The *nodes* keyword specifies the max number of compute nodes, and the *iprange* specifies the range of IP addresses that are assigned to those compute nodes.

By default and in general practice, node numbers and IP addresses are assigned to the compute nodes in the order that their *node* entries appear in the config file, beginning with node 0 and the first IP address specified by the *iprange* entry in the config file. For example, the config file entries:

```
nodes 8
iprange 10.20.30.100 10.20.30.107
node 00:01:02:03:04:1A 00:01:02:03:05:2A
node 00:01:02:03:04:1B 00:01:02:03:05:2B
node 00:01:02:03:04:1C 00:01:02:03:05:2B
node 00:01:02:03:04:1D 00:01:02:03:05:2B
```

specify a network that contains a maximum of eight nodes, with four nodes currently known, and with an IP address range that falls between the 10.20.30.100 lowerbound and the 10.20.30.107 upperbound. Here the node with MAC address `00:01:02:03:04:1C` is node 2 and will be assigned an IP address 10.20.30.102.

ClusterWare treats the upperbound IP address as optional, so all that is necessary to specify is:

```
nodes 8
iprange 10.20.30.100
```

and ClusterWare calculates the upperbound IP address. This is especially useful when dealing with large *nodes* counts, e.g.:

```
nodes 1357
iprange 10.20.30.100
```

when it becomes increasingly clumsy for the cluster administrator to accurately calculate the upperbound address.

An optional node number can explicitly specify an override node number:

```
node 00:01:02:03:04:1A 00:01:02:03:05:2A
node 00:01:02:03:04:1B 00:01:02:03:05:2B
node 2 00:01:02:03:04:1C 00:01:02:03:05:2B
node 00:01:02:03:04:1D 00:01:02:03:05:2B
```

explicitly (and redundantly) specifies the node 2 numbering. Alternatively:

```
node 00:01:02:03:04:1A 00:01:02:03:05:2A
node 00:01:02:03:04:1B 00:01:02:03:05:2B
node 5 00:01:02:03:04:1C 00:01:02:03:05:2B
node 00:01:02:03:04:1D 00:01:02:03:05:2B
```

explicitly names that node as node 5 with IP address 10.20.30.105, and the next node (with MAC address `00:01:02:03:04:1D` will now be node 6 with IP address 10.20.30.106.

In another variation, commenting-out the MAC address(es) leaves a node numbering gap for node 2, and MAC address `00:01:02:03:04:1D` continues to be known as node 3:

```
node 00:01:02:03:04:1A 00:01:02:03:05:2A
node 00:01:02:03:04:1B 00:01:02:03:05:2B
node # 00:01:02:03:04:1C 00:01:02:03:05:2B
node 00:01:02:03:04:1D 00:01:02:03:05:2B
```

However, if the node with that commented-out MAC address `00:01:02:03:04:1C` does attempt to PXE boot, then **beoserv** assigns a new node number (4) to that physical node and automatically appends a new *node* entry to the list (assuming the *nodeassign* policy is *append*, and assuming the *iprange* and *nodes* entries allow room for expansion). This appending results in:

```
node 00:01:02:03:04:1A 00:01:02:03:05:2A
node 00:01:02:03:04:1B 00:01:02:03:05:2B
node # 00:01:02:03:04:1C 00:01:02:03:05:2B
node 00:01:02:03:04:1D 00:01:02:03:05:2B
node 00:01:02:03:04:1C 00:01:02:03:05:2B
```

If you want to have **beoserv** ignore that physical node and keep the remaining nodes numbered without change, then use the keyword *off*:

```
node 00:01:02:03:04:1A 00:01:02:03:05:2A
node 00:01:02:03:04:1B 00:01:02:03:05:2B
node off 00:01:02:03:04:1C 00:01:02:03:05:2B
node 00:01:02:03:04:1D 00:01:02:03:05:2B
```

A *node* entry can identify itself as a non-Scyld node and can direct **beoserv** to respond to the node in a variety of ways, including telling the node to boot from a local harddrive, or provisioning the node with specific kernel and initrd images. See Chapter 7 for details.

### *Caching Shared Libraries*

To add a shared library to the list of libraries cached on the compute nodes, specify the pathname of the individual file or the pathname of the entire directory in which the file resides using the *libraries* directive. An open() syscall on a compute node to open a file thus named, or to open a file that resides in a named directory, will cause that file to be pulled from the master node to the compute node and saved in the local RAM filesystem.

The *prestage* directive names specific files to be pulled onto each compute node at node boot time. If a file pathname resides in one of the *libraries* directories, then BProc's filecache functionality pulls the file from the master node. Otherwise, the specified file is pushed from the master to the compute node at startup, with directories created as needed.

### Specifying node names and aliases

The *nodename* keyword in the master's `/etc/beowulf/config` affects the behavior of the ClusterWare NSS. Using the *nodename* keyword, one may redefine the primary host-name of the cluster, define additional hostname aliases for compute nodes, and define additional hostname (and hostname aliases) for entities loosely associated with the compute node's cluster position.

nodename [name-format] <IPv4 Offset or base> <netgroup>

The presence of the optional IPv4 argument defines if the entry is for "compute nodes" (i.e. the entry will resolve to the 'dot-number' name) or if the entry is for non-cluster entities that are loosely associated with the compute node. In the case where there **is** an IPv4 argument, the *nodename* keyword defines an additional hostname name that maps to an IPv4 address loosely associated with the node number. In case where IPv4 argument is present, the *nodename* keyword defines hostname and hostname aliases for the clustering interface (i.e. the compute nodes). Subsequent *nodename* entries without an IPv4 argument specify additional hostname aliases for compute nodes. In either case, the format string must contain a conversion specification for node number substitution. The conversion specification is introduced by a '*%*'. An optional following digit in the range 1..5 specifies a zero-padded minimum field width. The specification is completed with an 'N'. An unspecified or zero field width allows numeric interpretation to match compute node host names. For example, *n%N* will match *n23*, *n+23*, and *n0000023*. By contrast, *n%3N* will only match *n001* or *node023*, but not *n1* or *n23*.

### Compute node command-line options

The *kernelcommandline* directive is a method of passing various options to the compute node's kernel and to Beowulf on the node. There are a large number of different command line options that you can employ. This section covers some of them.

Some options are interpreted by the kernel on the compute node and ignored by Beowulf:

apic

This option turns on APIC support on the compute node. APIC is the newer of two different mechanisms Intel provides for invoking interrupts. It works better with SMP systems than the older mechanism, called XT-PIC. However, not every motherboard and chipset works correctly with APIC, so this option is disabled by default to avoid problems for those machines that do not support it.

If you find that your cluster nodes kernel panic or crash immediately upon boot, you probably want to turn off APIC by specifying *noapic* in the command line options. If you have many devices that generate interrupts (such as hard disk controllers, network adapters, etc.) you may want to try turning on APIC to see if there is any performance advantage for your cluster.

panic=<*seconds*>

This option allows you to specify how many seconds the kernel should wait to reboot after a kernel panic. For example, if you specify *panic=60*, then the kernel will wait 60 seconds before rebooting. Note that Beowulf automatically adds *panic=30* to final boot images.

apm=<*action*>

This option allows you to specify APM options on the compute node. Acceptable <*action*> values are *on* (to turn APM completely on), *off* (to turn it completely off), *debug* (to turn on debugging), and *power-off* (to turn on only the

power-off part of APM).

APM is not SMP-safe in the kernel; it will auto-disable itself if turned completely *on* for an SMP box. However, the *power-off* part of APM is SMP safe; thus, if you want to be able to power-off SMP boxes, you can do so by specifying *apm=power-off*. Note that *apm=power-off* is specified in the default *kernelcommandline* directive.

console=<*device*>, <*options*>

This option is used to select which device(s) to use for console output. For <*device*> use *ttyX* for the foreground virtual console, *ttyX* (e.g., *tty1*) for any other virtual console, and *ttySx* (e.g., *ttyS0* for a serial port.

For the serial port, <*options*> defines the baud rate/parity/bits of the port in the format "BBBBPN", where "BBBB" is the speed, "P" is parity (n/o/e), and "N" is bits. The default setting is *9600n8*, and the maximum baud rate is 115200. For example, to use the serial port at the maximum baud rate, specify *console=ttyS0,115200n8r*

Other options are interpreted by Beowulf on the compute node:

rootfs_size=<*size*>

A compute node employs a RAM-based root filesystem for local non-persistent storage, typically used to contain BProc's filecache libraries and other files, the `/tmp` directory, and other directories that are not mounted using some variety of global storage (e.g., NFS or Panfs) or on local harddrives. This *tmpfs* root filesystem consumes physical memory only as needed, which commonly is about 100- to 200-MBytes unless user workloads impose greater demands on (for example) `/tmp` space. However, by default the rootfs is allowed to grow to consume a maximum of 50% of physical memory, which has the potential of allowing users to consume (perhaps inadvertently) an excessive amount of RAM that would otherwise be available to applications' virtual memory needs.

This 50% default can be overridden by the judicious use of the <*size*> option, where <*size*> can be expressed as numeric bytes, megabytes (appending "m" or "M"), or gigabytes (appending "g" or "G"), or as a percentage of total physical memory (appending numeric value and "%"). Examples:

```
rootfs_size=2048m
rootfs_size=1G
rootfs_size=15%
```

Note that this override is rarely needed, and it must be utilized with care. An inappropriately constrained root filesystem will cripple the node, just as an inadequate amount of physical memory that is available for virtual memory will trigger Out-Of-Memory failures. The cluster administrator is encouraged to limit user filespace usage in other ways, such as declaring `/etc/security/limits.conf` limits on the max number of open files and/or the maximum filesize.

rootfs_timeout=<*seconds*>
getfile_timeout=<*seconds*>

The **beoclient** daemon on each compute node manages the early boot process, such as using tftp to read the kernel image and initrd files from the master node's **beoserv** daemon, and using tcp to read the initial root filesystem image (*rootfs*) from beoserv. After the node boots, BProc's filecache functionality on the compute node also uses tcp to read files from the master, as needed by applications.

The default timeout for these tcp reads is *30* seconds. If this timeout is too short, then add one of these options to the *kernelcommandline* to override the default. The option *getfile_timeout* overrides the timeout for all beoclient tcp read operations. The option *rootfs_timeout* overrides the timeout only for the tcp read of the root filesystem at node boot time.

syslog_server=<*IPaddress*>

By default, a compute node forwards its kernel messages and syslog messages back to the master node's **syslog** or **rsyslog** service, which then appends these log messages to the master's `/var/log/messages` file. Alternatively,

the cluster administrator may choose to instead forward these compute node log messages to another server by using the *syslog_server* option to identify the <*IPaddress*> of that server. This should be an IPv4 address, e.g., *syslog_server=10.20.30.2*.

Scyld ClusterWare automatically configures the master node's log service to handle incoming log messages from remote compute nodes. However, the cluster administrator must manually configure the alternate syslog server:

For the **syslog** service (Scyld ClusterWare 4 and 5), edit `/etc/sysconfig/syslog` on the alternate server to add "-r -x" to the variable SYSLOGD_OPTIONS.

For the **rsyslog** service (Scyld ClusterWare 6), edit `/etc/sysconfig/rsyslog` on the alternate server to add "-x" to the variable SYSLOGD_OPTIONS, and edit `/etc/rsyslog.conf` to un-comment the following lines to expose them, i.e., just as Scyld ClusterWare has done in the master node's `/etc/rsyslog.conf` file:

```
$ModLoad imudp.so
$UDPServerRun 514
```

Finally, restart the service on both the master node and the alternate syslog server before restarting the cluster.

### *Specifying kernel modules for use on compute nodes*

Each *bootmodule* entry identifies a kernel module to be added to the initrd that is passed to each compute node at boot time. These entries typically name possible Ethernet drivers used by nodes supplied by Penguin Computing. If the cluster contains nodes not supplied by Penguin Computing, then the cluster administrator should examine the default list and add new *bootmodule* entries as needed.

At boot time, Beowulf scans the node's PCI bus to determine what devices are present and what driver is required for each device. If the specified driver is named by a *bootmodule* entry, then Beowulf loads the module and all its dependencies. However, some needed modules are not found by this PCI scan, e.g., those used to manage specific filesystem types. These modules require adding an additional `config` file entry: *modprobe*. For example:

```
modprobe xfs
```

Note that each named *modprobe* module must also be named as a *bootmodule*.

You may also specify module-specific arguments to be applied at module load time, e.g.,

```
modarg forcedeth optimization_mode=1
```

RHEL7 introduced externally visible discrete firmware files that are associated with specific kernel software drivers. When **modprobe** attempts to load a kernel module that contains such a software driver, and that driver determines that the controller hardware needs one or more specific firmware images (which are commonly found in `/lib/firmware`), then the kernel first looks at its list of built-in firmware files. If the desired file is not found in that list, then the kernel sends a request to the **udevd** daemon to locate the file and to pass its contents back to the driver, which then downloads the contents to the controller. This functionality is problematic if the kernel module is an `/etc/beowulf/config` *bootmodule* and is an Ethernet driver that is necessary to boot a particular compute node in the cluster. The number of `/lib/firmware/` files associated with every possible *bootmodule* module is too large to embed into the `initrd` image common to all compute nodes, as that burdens every node with a likely unnecessarily oversized `initrd` to download. Accordingly, the cluster administrator must determine which specific firmware file(s) are actually required for a particular cluster and are not yet built-in to the kernel, then add *firmware* directive(s) for those files.

A *bootmodule* firmware problem exhibits itself as a compute node which does not boot because the needed Ethernet driver cannot be **modprobe**'d because it cannot load a specific firmware file. After a timeout waiting for **udevd** to unsuccessfully find the file, the compute node typically reboots - endlessly, as it continues to be unable to load the needed firmware file.

The cluster administrator can use the *firmware* directive to add specific firmware files to the compute node `initrd`, as needed. The compute node kernel writes the relevant firmware filename information to its console, e.g. a line of the form:

```
Failed to load firmware "bnx2/bnx2-mips-06-6.2.1.fw"
```

Ideally, the administrator gains access to the node's console to see the specific filename, then adds a directive to `/etc/beowulf/config`:

```
firmware bnx2/bnx2-mips-06-6.2.1.fw
```

and rebuilds the initrd:

```
[root@cluster ~] # service clusterware reload
```

(Note: *reload*, not *restart*)

If the node continues to fail to boot, then the failure is likely due to another missing firmware file. Check the node's console output again, and add the specified file to the *firmware* directive.

If the cluster administrator cannot easily see the node's console output to determine what firmware files are needed, then if the administrator knows the likely *bootmodule* module culprit, then the administrator can brute-force every known firmware file for that module using a directive of the form:

```
firmware bnx2
```

that names an entire `/lib/firmware/` subdirectory. This will likely create a huge `initrd` that will (if the correct *bootmodule* module is specified) successfully boot the compute node. The administrator should then examine the node's syslog output, which is typically seen in `/var/log/messages`, to determine the specific individual firmware filenames that were actually needed, and then the administrator replaces the subdirectory name with the now-known specific firmware filenames. Subsequently, the cluster administrator should contact Penguin Computing Support to inform us what those needed firmware files are, so that we can build-in these files into future kernel images and thus allow the cluster administrator to remove the *firmware* directives and thus reduce the `initrd` size, which contains not only the firmware images, but additionally includes various executable binaries and libraries that are only needed for this dynamic **udevd** functionality.

## /etc/beowulf/fdisk

The `/etc/beowulf/fdisk` directory is created by the **beofdisk** utility when it evaluates local disks on individual compute nodes and creates partition tables for them. For each unique drive geometry discovered among the local disks on the compute nodes, **beofdisk** creates a file within this directory. The file naming convention is "head;ccc;hhh;sss", where "ccc" is the number of cylinders on the disk, "hhh" is the number of heads, and "sss" is the number of sectors per track.

These files contain the partition table information as read by **beofdisk**. Normally, these files should not be edited by hand.

You may create separate versions of this directory that end with the node number (for example, `/etc/beowulf/fdisk.3`). The master's **BeoBoot** software will look for these directories before using the general `/etc/beowulf/fdisk` directory.

For more information, see the section on **beofdisk** in the *Reference Guide*.

## /etc/beowulf/fstab

This is the filesystem table for the mount points of the partitions on the compute nodes. It should be familiar to anyone who has dealt with an `/etc/fstab` file in a standard Linux system, though with a few Scyld ClusterWare extensions. For details, see the *Reference Guide* or execute **man beowulf-fstab**.

You may create separate node-specific versions by appending the node number, e.g., `/etc/beowulf/fstab.3` for node 3. The master's beoboot **node_up** script looks first for a node_specific `fstab.N` file, then if no such file exists will use the default `/etc/beowulf/fstab` file.

> **Caution**
>
> On compute nodes, NFS directories must be mounted using either the IP address or the *$MASTER* keyword; the master node's hostname cannot be used. This is because `/etc/beowulf/fstab` is evaluated before the Scyld ClusterWare name service is initialized, which means hostnames cannot be resolved on a compute node at that point.

### /etc/beowulf/backups/

This directory contains time-stamped backups of older versions of various configuration files, e.g., `/etc/beowulf/config` and `/etc/beowulf/fstab`, to assist in the recovery of a working configuration after an invalid edit.

### /etc/beowulf/conf.d/

This directory contains various configuration files that are involved when booting a compute node. In particular, the **node_up** script pushes the master node's `/etc/beowulf/conf.d/limits.conf` to each compute node as `/etc/security/limits.conf`, and pushes `/etc/beowulf/conf.d/sysctl.conf` to each compute node as `/etc/sysctl.conf`. If `/etc/beowulf/conf.d/limits.conf` does not exist, then **node_up** creates an initial file as a concatenation of the master node's `/etc/security/limits.conf` plus all files in the directory `/etc/security/limits.d/`. Similarly, **node_up** creates an initial `/etc/beowulf/conf.d/sysctl.conf` (if it doesn't already exist) as a copy of the master's `/etc/sysctl.conf`. The cluster administrator may subsequently modify these initial "best guess" configuration files as needed for compute nodes.

## Command Line Tools

### bpstat

The command **bpstat** can be used to quickly check the status of the cluster nodes and/or see what processes are running on the compute nodes. See the *Reference Guide* for details on usage.

### bpctl

To reboot or set the state of a node via the command line, one can use the **bpctl** command. For example, to reboot node 5:

```
[root@cluster ~] # bpctl -S 5 -R
```

As the administrator, you may at some point have reason to prevent other users from running new jobs on a specific node, but you do not want to shut it down. For this purpose we have the *unavailable* state. When a node is set to *unavailable* non-root users will be unable to start new jobs on that node, but existing jobs will continue running. To do this, set the state to *unavailable* using the **bpctl** command. For example, to set node 5 to *unavailable*:

```
[root@cluster ~] # bpctl -S 5 -s unavailable
```

### node_down

If you are mounting local filesystems on the compute nodes, you should shut down the node cleanly so that the filesystems on the harddrives stay in a consistent state. The **node_down** script in `/usr/lib/beoboot/bin` does exactly this. It takes two arguments; the first is the node number, and the second is the state to which you want the node to go. For example, to cleanly reboot node 5:

```
[root@cluster ~] # /usr/lib/beoboot/bin/node_down 5 reboot
```

Alternatively, to cleanly power-off node 5:

```
[root@cluster ~] # /usr/lib/beoboot/bin/node_down 5 pwroff
```

The **node_down** script works by first setting the node's state to unavailable, then remounting the filesystems on the compute node read-only, then calling **bpctl** to change the node state. This can all be done by hand, but the script saves some keystrokes.

To configure **node_down** to use IPMI, set the *ipmi* value in `/etc/beowulf/config` to *enabled* as follows:

```
[root@cluster ~] # beoconfig ipmi enabled
```

## Configuring CPU speed/power for Compute Nodes

Modern motherboards and processors support a degree of administrator management of CPU frequency within a range defined by the motherboard's BIOS. Scyld ClusterWare provides the `/etc/beowulf/init.d/30cpuspeed` script and its associated `/etc/beowulf/conf.d/cpuspeed.conf` configuration file to implement this management for compute nodes. The local cluster administrator is encouraged to review the `cpuspeed.conf` config file's section labeled *Scaling governor values* and potentially adjust the environment variable SCALINGGOV as desired, and then to enable the `30cpuspeed` script:

```
[root@cluster ~] # beochkconfig 30cpuspeed on
```

The administrator should also ensure that no other *cpuspeed* or *cpupower* script is enabled for compute nodes.

In brief, the administrator can choose among four CPU scaling governor settings:

- *performance*, which directs the CPUs to execute at the maximum frequency supported by the motherboard and processor, as specified by the motherboard BIOS.
- *powersave*, which directs the CPUs to execute at the minimum frequency supported by the motherboard and processor.
- *ondemand*, which directs the kernel to adjust the CPU frequency between the minimum and maximum. An idle CPU executes at the minimum. As a load appears, the frequency increases relatively quickly to the maximum, and if and when the load subsides, then the frequency decreases back to the minimum. This is the default setting.
- *conservative*, which similarly directs the kernel to adjust the CPU frequency between the minimum and maximum, albeit making those adjustments with somewhat longer latency than is done for *ondemand*.

The upside of the *performance* scaling governor is that applications running on compute nodes always enjoy the maximum CPU frequencies that are supported by the node hardware. The downside is that even idle CPUs consume that same maximum power and thus generate maximum heat. For the scaling governors *performance*, *ondemand*, and *conservative*, a computebound workload drives the CPU frequencies (and power and heat) to the maximum, and thus computebound application performance will exhibit little or no difference among those governors. However, a workload of rapid context

switching and frequent idle time will show perhaps 10-20% lower performance for *ondemand* versus *performance*, and possibly an even larger decline with *conservative*. The *powersave* governor is typically only employed when a need to minimize the cluster power consumption and/or minimize thermal levels outweighs a need to achieve maximum performance.

A broader discussion can be found in the `/usr/share/doc/kernel-doc-2.6.32/Documentation/cpu-freq/` documents, e.g., `governors.txt`. Install the RHEL7 or CentOS7 base distribution's *kernel-doc* package to access these documents.

## Adding New Kernel Modules

The **modprobe** command uses `/usr/lib/`\`uname -r\`/`modules.dep.bin` to determine the pathnames of the specified kernel module and that module's dependencies. The **depmod** command builds the human-readable `modules.dep` and the binary `module.dep.bin` files, and it should be executed *on the master node* after installing any new kernel module.

Executing **modprobe** on a compute node requires additional caution. The first use of **modprobe** retrieves the current `modules.dep.bin` from the master node using bproc's *filecache* functionality. Since any subsequent **depmod** on the master node rebuilds `modules.dep.bin`, then a subsequent **modprobe** on a compute node will only see the new `modules.dep.bin` if that file is copied to the node using **bpcp**, or if the node is rebooted and thereby silently retrieves the new file.

In general, you should not execute **depmod** on a compute node, since that command will only see those few kernel modules that have previously been retrieved from the master node, which means the node's newly built `modules.dep.bin` will only be a sparse subset of the master node's full `module.dep.bin`. Bproc's *filecache* functionality will always properly retrieve a kernel module from the master node, as long as the node's `module.dep.bin` properly specifies the pathname of that module, so the key is to have the node's `module.dep.bin` be a current copy of the master's file.

Many device drivers are included with Scyld ClusterWare and are supported out-of-the-box for both the master and the compute nodes. If you find that a device, such as your Ethernet adapter, is not supported and a Linux source code driver exists for it, then you will need to build the driver modules for the master.

To do this, you will need to install the RPM of kernel source code (if you haven't already done so). Next, compile the source code using the following extra GNU C Compiler (gcc) options.

```
-D__BOOT_KERNEL_SMP=1 -D__BOOT_KERNEL_UP=0
```

The compiled modules must be installed in the appropriate directories under `/lib/modules`. For example, if you are currently running under the 2.6.9-67.0.4.ELsmp kernel version, the compiled module for an Ethernet driver would be put in the following directory:

```
/lib/modules/2.6.9-67.0.4.ELsmp/kernel/drivers/net
```

Any kernel module that is required to boot a compute node, e.g., most commonly the Ethernet driver(s) used by compute nodes, needs special treatment. Edit the config file `/etc/beowulf/config` to add the name of the driver to the *bootmodule* list; you can add more *bootmodule* lines if needed. See the Section called *Specifying kernel modules for use on compute nodes* for more details.

Next, you need to configure how the device driver gets loaded. You can set it up so that the device driver only loads if the specific device is found on the compute node. To do this, you need to add the PCI vendor/device ID pair to the PCI table information in the `/usr/share/hwdata/pcitable` file. You can figure out what these values are by using a combination of **lspci** and **lspci -n**.

So that your new kernel module is always loaded on the compute nodes, include the module in the initial RAM disk by adding a *modprobe* line to `/etc/beowulf/config`. The line should look like the following:

```
modprobe <module>
```

where <*module*> is the kernel module in question.

Finally, you can regenerate the **BeoBoot** images by running **service clusterware reload**. For more details, see the Section called *Changing Boot Settings* in Chapter 10.

## Accessing External License Servers

To configure the firewall for accessing external license servers, enable **ipforward** in the `/etc/beowulf/config` file. The line should read as follows:

```
ipforward yes
```

You must then reboot the compute nodes and restart the cluster services. To do so, run the following two commands as root in quick succession:

```
[root@cluster ~] # bpctl -S all -R
[root@cluster ~] # service clusterware restart
```

> **Tip:** If IP forwarding is enabled in `/etc/beowulf/config` but is still not working, then check `/etc/sysctl.conf` to see if it is disabled.
>
> Check for the line "net.ipv4.ip_forward = 1". If the value is set to 0 (zero) instead of 1, then IP forwarding will be disabled, even if it is enabled in `/etc/beowulf/config`.

## Configuring SSH for Remote Job Execution

Most applications that leverage **/usr/bin/ssh** on compute nodes can be configured to use **/usr/bin/rsh**. In the event that your application requires SSH access to compute nodes, ClusterWare provides this ability through `/etc/beowulf/init.d/81sshd`. To start **/usr/sbin/sshd** on compute nodes, enable the `81sshd` script and reboot your nodes:

```
[root@cluster ~] # beochkconfig 81sshd on
[root@cluster ~] # bpctl -S all -R
```

When each node boots, `81sshd` starts **sshd** on the node, and the master's root user will be able to SSH to a compute node without a password, e.g.:

```
[root@cluster ~] # ssh n0 ls
```

By default, compute node **sshd** daemons do not allow for password-based authentication -- only key-based authentication is available -- and only the root user's SSH keys have been configured.

If a non-root user needs SSH access to compute nodes, the user's SSH keys will need to be configured. For example, create a DSA key using ssh-keygen, and hit *Enter* when prompted for a password if you want password-less authentication:

```
[user1@cluster ~] $ ssh-keygen -t dsa
```

Since the master's `/home` directory is mounted (by default) as `/home` on the compute nodes, just copy the public key to ~/.ssh/authorized_keys:

```
[user1@cluster ~] $ cp -a ~/.ssh/id_dsa.pub ~/.ssh/authorized_keys
```

Now the user can run commands over SSH to any node using shared key authentication:

```
[user1@cluster ~] $ ssh n0 date
```

If you wish to modify **sshd**'s settings, you can edit `/etc/beowulf/conf.d/sshd_config` and then reboot the nodes. Node-specific **sshd** configuration settings can be saved as `/etc/beowulf/conf.d/sshd_config.$NODE`.

Client behavior for SSH on the nodes can be adjusted by editing the global `/etc/beowulf/conf.d/ssh_config` or a node-specific `/etc/beowulf/conf.d/ssh_config.$NODE`. This SSH client configuration will only be useful when using SSH from node to node. For example:

```
[user1@cluster ~] $ ssh n0 ssh n1 ls
```

Note that `/etc/beowulf/conf.d/sshd_config` and `ssh_config` only affect SSH behavior on compute nodes. The master's SSH configuration will not be affected.

## Interconnects

There are many different types of network fabric one can use to interconnect the nodes of your cluster. The least expensive and most common is Fast (100Mbps) and Gigabit (1000Mbps) Ethernet. Other cluster-specific network types, such Infiniband, offer lower latency, higher bandwidth and features such as RDMA (Remote Direct Memory Access).

### Ethernet

Switching fabric is always the most important (and expensive) part of any interconnected sub-system. Ethernet switches with up to 48 ports are extremely cost effective; however, anything larger becomes expensive quickly. Intelligent switches (those with software monitoring and configuration) can be used effectively to partition sets of nodes into separate clusters using VLANs; this allows nodes to be easily reconfigured between clusters if necessary.

#### *Adding a New Ethernet Driver*

Drivers for most Ethernet adapters are included with the Linux distribution, and are supported out of the box for both the master and the compute nodes. If you find that your card is not supported, and a Linux source code driver exists for it, you need to compile it against the master's kernel, and then add it to the cluster config file using the *bootmodule* keyword. See the *Reference Guide* for a discussion on the cluster config file.

For details on adding new kernel modules, see Adding New Kernel Modules earlier in this section.

#### *Gigabit Ethernet vs. Specialized Cluster Interconnects*

Surprisingly, the packet latency for Gigabit Ethernet is approximately the same as for Fast Ethernet. In some cases, the latency may even be slightly higher, as the network is tuned for high bandwidth with low system impact utilization. Thus Gigabit Ethernet will not give significant improvement over Fast Ethernet to fine-grained communication-bound parallel applications, where specialized interconnects have a significant performance advantage.

However, Gigabit Ethernet can be very efficient when doing large I/O transfers, which may dominate the overall run-time of a system.

## Other Interconnects

Infiniband is a new, standardized interconnect for system area networking. While the hardware interface is an industry standard, the details of the hardware device interface are vendor specific and change rapidly. Contact Scyld Customer Support for details on which Infiniband host adapters and switches are currently supported.

With the exception of unique network monitoring tools for each, the administrative and end user interaction is unchanged from the base Scyld ClusterWare system.

# Chapter 4. Remote Administration and Monitoring

Scyld ClusterWare provides a variety of tools for remotely monitoring and administering clusters. These include traditional shell and X window based tools, along with web based tools. Some utilities are available for users to monitor the system, while others are for administrators to configure the cluster.

## Command Line Tools

Scyld ClusterWare includes **openssh** by default. This tool allows you to securely log into your master node remotely and use the command line tools to manually edit the configuration files. For more information on the configuration files and command line utilities, see the Section called *Configuring the Cluster Manually* in Chapter 3.

## X Forwarding

SSH can also be configured to do X forwarding, which allows the GUI applications to be run on a remote machine. This allows you to use the full functionality of the convenient graphics tools, but can be slow, especially if the connection to the cluster is not via a local area network. In order to activate X forwarding, you may need to use the -X option to ssh from your client. Once the X forwarding is setup, you can use any of the GUI tools described throughout this manual.

# Chapter 5. Managing Users on the Cluster

In order for someone to gain access to a cluster, s/he must first be given a user account. The cluster administrator can manage user accounts with the same tools that are available with most Linux distributions. User access to cluster resources can also be controlled by the cluster administrator.

This chapter discusses the tools and commands for managing user accounts and controlling access to cluster resources.

## Managing User Accounts

### Adding New Users

The **useradd** command enables you to add a new user to the system. This command takes a single argument, which is the new user's login name:

```
[root@cluster ~] # useradd <username>
```

This command also creates a home directory named `/home/<username>`.

After you add the user, give them a default password using the **passwd** command so that they will be able to log in. This command takes a single argument, which is the username:

```
[root@cluster ~] # passwd <username>
```

**Tip:** It is good practice to give each user their own unique home directory.

### Removing Users

To remove a user from your cluster, use the **userdel** command. This command takes a single argument, which is the username:

```
[root@cluster ~] # userdel <username>
```

By default, **userdel** does not remove the user's home directory. To remove the home directory, include the **-r** option in the command:

```
[root@cluster ~] # userdel -r <username>
```

**Tip:** The **userdel** command will never remove any files that are not in the user's home directory. To fully remove all of a user's files, remove the user's mail file from `/var/spool/mail/`, as well as any files the user may have in `/tmp/`, `/var/tmp/`, and any other directories to which the user had write permissions.

# Managing User Groups

In addition to user accounts, you can also create user groups. Groups can be very powerful, as they allow you to assign resources to an arbitrary set of users. Groups are typically used for file permissions. However, you can also utilize groups to assign nodes to a specific set of users, thereby limiting which users have access to certain nodes. This section covers creating and modifying groups.

## Creating a Group

Before you can add users to a group, you must first create the group. Groups can be created with the **groupadd** command. This command takes a single argument, which represents the name of the group:

```
[root@cluster ~] # groupadd <groupname>
```

## Adding a User to a Group

Use the **usermod** command To add a user to a group. This command requires you to list all the groups the user should be a member of. To avoid accidentally removing any of the user's groups, first use the **groups** command to get a list of the user's current groups. The following example shows how to find the groups for a user named Smith:

```
[root@cluster ~] # groups smith
smith : smith src
```

After getting a list of the user's current groups, you can then add them to new groups, for example:

```
[root@cluster ~] # usermod -G smith,src,<newgroup> smith
```

## Removing a Group

To remove a group, run the **groupdel** command with the groupname as an argument:

```
[root@cluster ~] # groupdel <groupname>
```

# Controlling Access to Cluster Resources

By default, anyone who can log into the master node of the cluster can send a job to any compute node. This is not always desirable. You can use *node ownership* and *mode* to restrict the use of each node to a certain user or group, including restricting compute node access to the master node.

## What Node Ownership Means

Each node (including the master node) has *user*, *group* and *mode* bits assigned to it; these indicate who is allowed to run jobs on that node. The *user* and *group* bits can be set to any user ID or group ID on your system. In addition, the use of a node can be unrestricted by setting the *user* and *group* to "root".

For the **BProc** unified process space, the node permissions "root" and "any" are equivalent. Node user access follows the normal Linux convention, i.e., the most restrictive access rule is the one used. Some examples:

- user "root", group "test", mode 101 (u=1, g=0, o=1) — Users in the group "test" will not be able to access the node.
- user "tester", group "root", and mode 011 (u=0, g=1, o=1) — The user "tester" will not be able to access the node.
- user "tester", group "test", and mode 110 (u=0, g=1, o=1) — The user "tester" and users in the group "test" are the only non-root users able to access the node.

**Tip:** In Linux systems, "other" is defined as anyone not listed in the user or group.

## Checking Node Ownership

Display the current node access state by running the **bpstat** command:

```
[root@cluster ~] # bpstat -M
Node(s)  Status  Mode       User        Group
16-31    down    ---------- root        root
-1       up      ---x--x--x root        root
0-15     up      ---x--x--x root        root
```

The "User" column shows the user for each node and the "Group" column shows the group for each node. This display shows a cluster with default access permissions.

## Setting Node Ownership

You can set node ownership with the **bpctl** command. Use the **-S** option to specify which node to change. Use either the **-u** option to change the user, **-g** option to change the group, or **-m** to change the mode. The only bit utilized for the mode is the *execute* bit. Following are some examples.

- The following sets the user for node 5 to *root*:

  ```
  [root@cluster ~] # bpctl -S 5 -u root
  ```

- The following sets all the compute nodes to be in the group *beousers*:

  ```
  [root@cluster ~] # bpctl -S all -g beousers
  ```

- The following allows only the group *beousers* to access the compute nodes:

  ```
  [root@cluster ~] # bpctl -S all -m 010 -g beousers
  ```

- The following disallows non-root users to execute on the master:

  ```
  [root@cluster ~] # bpctl -M -m 0110
  ```

For example:

```
[root@cluster ~] # bpctl -M -m 0110
[root@cluster ~] # bpctl -S 0-3 -g physics
[root@cluster ~] # bpstat -M
Node(s)  Status  Mode       User        Group
16-31    down    ---------- root        root
-1       up      ---x--x--- root        root
0-3      up      ---x--x--x root        physics
4-15     up      ---x--x--x root        root
```

See the *Reference Guide* for additional details on **bpctl**.

Using **bpctl** does not permanently change the node ownership settings. Whenever the master node reboots or **service clusterware restart** reboots the cluster, the node ownership settings revert to the default of full, unrestricted access, or to the optional override settings specified by the *nodeaccess* directive(s) in the `/etc/beowulf/config` file. To make permanent changes to these settings, you must edit this file. For example, to make the above setting persistent, add the *nodeaccess* entries:

```
nodeaccess -M -m 0110
nodeaccess -S 0-3 -g physics
```

The *Reference Guide* and **man beowulf-config** provides details for the `/etc/beowulf/config` file.

# Chapter 6. Job Batching

For Scyld ClusterWare HPC, the default installation includes both the TORQUE resource manager and the Slurm workload manager, each providing users with an intuitive interface for remotely initiating and managing batch jobs on distributed compute nodes.

ClusterWare TORQUE is a customized redistribution of Open Source software that derives from Adaptive Computing Enterprises, Inc.[1] TORQUE is an Open Source tool based on standard OpenPBS. Scyld Slurm is a redistribution of Open Source software that derives from http://slurm.schedmd.com/, and the associated Munge package derives from http://dun.github.io/munge/.

Both TORQUE and Slurm are installed by default, although only one job manager can be enabled at any one time. See the Section called *Enabling TORQUE or Slurm*, below, for details. See the Scyld ClusterWare HPC *User's Guide* for general information about using TORQUE or Slurm. See Chapter 8 for details about how to configure TORQUE for high availability using multiple master nodes.

Scyld also redistributes the Scyld Maui job scheduler, also derived from Adaptive Computing, that functions in conjunction with the TORQUE job manager. The alternative Moab job scheduler is also available from Adaptive Computing with a separate license, giving customers additional job scheduling, reporting, and monitoring capabilities.

In addition, Scyld provides support for most popular open source and commercial schedulers and resource managers, including SGE, LSF, and PBSPro. For the latest information, see the Penguin Computing Support Portal at http://www.penguincomputing.com/support.

## Enabling TORQUE or Slurm

To enable TORQUE, then after all compute nodes are up and running, you must first disable SLURM, then enable and configure TORQUE, then reboot all the compute nodes:

```
slurm-scyld.setup cluster-stop
beochkconfig 91slurm off
slurm-scyld.setup disable
beochkconfig 90torque on
torque-scyld.setup reconfigure      # when needed
torque-scyld.setup enable
torque-scyld.setup cluster-start
torque-scyld.setup status
bpctl -S all -R
```

To enable Slurm, then after all compute nodes are up and running, you must first disable TORQUE, then enable and configure Slurm, then reboot all the compute nodes:

```
torque-scyld.setup cluster-stop
beochkconfig 90torque off
torque-scyld.setup disable
beochkconfig 91slurm on
slurm-scyld.setup reconfigure      # when needed
slurm-scyld.setup enable
slurm-scyld.setup cluster-start
slurm-scyld.setup status
bpctl -S all -R
```

Note: slurmdbd uses mariadb to create a database defined by `/etc/slurm/slurmdbd.conf`, and expects mariadb to be configured with no password.

Each Slurm user must setup the PATH and LD_LIBRARY_PATH environment variables to properly access the Slurm commands. This is done automatically for users who login when the Slurm service is running and TORQUE is not running. Alternatively, each Slurm user can manually execute **module load slurm** or can add that command line to (for example) the user's `.bash_profile`.

## Notes

1. http://www.adaptivecomputing.com/
2. http://slurm.schedmd.com/
3. http://dun.github.io/munge/
4. http://www.penguincomputing.com/support

# Chapter 7. Managing Non-Scyld Nodes

A ClusterWare cluster typically consists of a Scyld master node and one or more Scyld compute nodes, integrated and communicating across the private cluster network interface. However, ClusterWare also supports additional devices and nodes that may reside on that private cluster network. This section describes how these Scyld and non-Scyld nodes are configured using entries in the `/etc/beowulf/config` file.

## DHCP IP address assignment to devices

The private cluster network may have one or more devices attached to it that issue a DHCP request to obtain a dynamic IP address, vs. the device being configured with a static IP address. Typically, only the master node (or nodes - see Chapter 8) owns a static IP address.

> **Caution**
>
> Care must be taken with static IP addresses to guarantee there are no address collisions.

Examples of such devices are managed switches and storage servers. The **beoserv** DHCP service for such devices is configured using the *host* directive, together with an associated *hostname* directive. For example,

```
nodes 32
iprange 10.20.30.100 10.20.30.131 # IPaddr range of compute nodes
...
hostrange 10.20.30.4  10.20.30.9 # IPaddr range of devices for DHCP
hostrange 10.20.30.90 10.20.30.99 # IPaddr range of PDUs for DHCP
...
host 00:A0:D1:E9:87:CA 10.20.30.5 smartswitch
host 00:A0:D1:E3:FC:E2 10.20.30.90 pdu1
host 00:A0:D1:E3:FD:4A 10.20.30.91 pdu2
```

The *host* keyword affects both the beoserv DHCP server and how the ClusterWare NSS responds to hostname lookups. The *host* keyword associates a non-cluster entity, identified by its MAC address, to an IP address that should be delivered to that client entity, if and when it makes a DHCP request to the master node, together with one or more optional hostnames to be associated with this IP address.

If the hostname is provided, then normal NSS functionality is available. Using the above example, then:

```
[user1@cluster ~] $ getent hosts smartswitch
```

returns:

```
10.20.30.5 smartswitch
```

and

```
[user1@cluster ~] $ getent ethers 00:A0:D1:E9:87:CA
```

returns:

```
00:a0:d1:e9:87:ca smartswitch
```

Each *host* IP address must fall within a defined *hostrange* range of IP addresses. Moreover, each of the potentially multiple *hostrange* ranges must not overlap any other range, must not overlap the cluster compute nodes range that is defined by the *iprange* directive, and must not collide with IP address(es) of master node(s) on the private network.

## Simple provisioning using PXE

A default *node* entry, such as:

```
node 00:A0:D1:E5:C4:6E 00:A0:D1:E5:C4:6F
```

or an explicitly numbered *node* entry, such as one for node15:

```
node 15 00:A0:D1:E5:C4:6E 00:A0:D1:E5:C4:6F
```

is assumed to be a Scyld node, and a PXE request from one of these MAC addresses results in **beoserv** provisioning the node with the kernel image, initrd image, and kernel command-line arguments that are specified in `/etc/beowulf/config` file entries, e.g.:

```
kernelimage /boot/vmlinuz-2.6.18-164.2.1.el5.540g0000
initrdimage /var/beowulf/boot/computenode.initrd
kernelcommandline rw root=/dev/ram0 image=/var/beowulf/boot/computenode.rootfs
```

ClusterWare automatically maintains the `config` file's default *kernelimage* to specify the same kernel that currently executes on the master node. A Scyld node integrates into the **BProc** unified process space.

Enhanced syntax allows for custom booting of different kernel and initrd images. For example, specific nodes can boot a standalone RAM memory test in lieu of booting a full Linux kernel:

```
kernelimage 15 /var/beowulf/boot/memtest86+-4.00.bin
initrdimage 15 none
kernelcommandline 15 none
```

Thus when node15 makes a PXE request, it gets provisioned with the specified binary image that performs a memory test. In the above example, the *initrdimage* of *none* means that no initrd image is provisioned to the node because that particular memory test binary doesn't need an initrd. Moreover, the node number specifier of *15* can be a range of node numbers, each of which would be provisioned with the same memory test.

## Simple provisioning using the *class* directive

An optional `config` file *class* directive assigns a name to a set of image and kernel command-line arguments. The previous example can be alternatively accomplished with:

```
class memtest kernelimage /var/beowulf/boot/memtest86+-4.00.bin
class memtest initrdimage none
class memtest kernelcommandline none
...
node 15 memtest 00:A0:D1:E5:C4:6E 00:A0:D1:E5:C4:6F
```

which results in the same memory test provisioning of node15 as seen earlier.

Similarly, the default Scyld node provisioning can be expressed as:

```
class scyld kernelimage /boot/vmlinuz-2.6.18-164.2.1.el5.540g0000
class scyld initrdimage /var/beowulf/boot/computenode.initrd
class scyld kernelcommandline rw root=/dev/ram0 image=/var/beowulf/boot/computenode.rootfs
...
node scyld pxe pxe 00:A0:D1:E5:C4:6E 00:A0:D1:E5:C4:6F
```

The first *pxe* is termed the **boot-sequence**, and the second *pxe* is termed the **boot-stage**. The **boot-stage** describes how **beoserv** should respond to a node's PXE request. In the example above, the **boot-stage** of *pxe* instructs **beoserv** to respond to the node's first PXE request with the kernel image, initrd image, and kernel command-line specified in the class *scyld*.

## Booting a node from the local harddrive

The *node* entry's **boot-sequence** and **boot-stage** have more powerful capabilities. For example, suppose node15 is installed with a full distribution of CentOS 4.8 on a local harddrive, and suppose the master node's `config` file contains entries:

```
class genericboot kernelimage none
class genericboot initrdimage none
class genericboot kernelcommandline none
...
node 15 genericboot pxe+local local 00:A0:D1:E5:C4:6E 00:A0:D1:E5:C4:6F
```

When node15 boots, it first makes a DHCP request to join the private cluster network, then it attempts to boot, abiding by the specific sequence of boot devices named in its BIOS. ClusterWare expects that the first boot device is PXE over Ethernet, and the second boot device is a local harddrive. When node15 initiates its PXE request to the master node, **beoserv** sees the **boot-stage** of *local* and thus directs node15 to "boot next", i.e., to boot from the local harddrive.

## Provisioning a non-Scyld node

In the previous example, we assumed that node15 already had a functioning, bootable operating system already installed on the node. Having a preexisting installation is not a requirement. Suppose the `config` file contains entries:

```
class centos5u4 kernelimage /var/beowulf/boot/vmlinuz-centos5u4_amd64
class centos5u4 initrdimage /var/beowulf/boot/initrd-centos5u4_amd64.img
class centos5u4 kernelcommandline initrd=initrd-centos5u4_amd64.img
                  ks=nfs:10.1.1.1:/home/os/kickstarts/n5-ks.cfg ksdevice=eth0
...
node 15 centos5u4 pxe+local pxe 00:A0:D1:E5:C4:6E 00:A0:D1:E5:C4:6F
```

(where the *kernelcommandline* has been broken into two lines for readability, although in reality it must be a single line in the `config` file). This time node15's PXE request arrives, and the **boot-stage** of *pxe* directs **beoserv** to respond with the *class centos5u4* kernel image, initrd image, and kernel command-line arguments. The latter's *ks* arguments informs node15's kernel to initiate a **kickstart** operation, which is a Red Hat functionality that provisions the requester with rpms and other configuration settings as specified in the `/home/os/kickstarts/n5-ks.cfg` kickstart configuration file found on the master node. It is the responsibility of the cluster administrator to create this kickstart file. See the Section called *Sample Kickstart Script* in Appendix A for a sample configuration file.

After this initial PXE response (i.e., the *pxe* step of the *pxe+local* **boot-sequence**), **beoserv** rewrites the *node* entry to change the **boot-stage** to the *local* second step of the *pxe+local* **boot-sequence**. For example,

```
node 15 centos5u4 pxe+local pxe 00:A0:D1:E5:C4:6E 00:A0:D1:E5:C4:6F
```

gets automatically changed to:

```
node 15 centos5u4 pxe+local local 00:A0:D1:E5:C4:6E 00:A0:D1:E5:C4:6F
```

What this accomplishes is: the first PXE request is met with a directive to boot a kernel on node15 that initiates the **kickstart** provisioning, and then any subsequent PXE request from node15 (presumably from a now-fully provisioned node) results in a **beoserv** directive to node15 to "boot next", i.e., to boot from the local harddrive.

If the cluster administrator wishes to reprovision the node and start fresh, then simply change the **boot-stage** from *local* back to *pxe*, and run **service clusterware reload** to instruct **beoserv** to re-read the config file to see your manual changes.

If you want the node to **kickstart** reprovision on every boot (albeit an unlikely scenario, but presented here for completeness), then you would configure this using:

```
node 15 centos5u4 pxe pxe 00:A0:D1:E5:C4:6E 00:A0:D1:E5:C4:6F
```

## Integrating a non-Scyld node into the cluster

A non-Scyld node that locally boots a full distribution operating system environment may have an assigned IP address in the private cluster network *iprange*, but it is initially invisible to the master node's monitoring tools and job manager. The **bpstat** tool only knows about Scyld nodes, and the more general **beostatus** is ignorant of the non-Scyld node's presence in the cluster. The non-Scyld node is itself ignorant about the names and IP addresses of other nodes in the cluster, whether they be Scyld or non-Scyld nodes, until and unless the cluster administrator adds each and every node into the non-Scyld node's local `/etc/hosts` file.

This shortcoming can be remedied by installing two special ClusterWare packages onto the non-Scyld node: **beostat-sendstats** and **beonss-kickbackclient**. These packages contain the client-side pieces of **beostat** and **beonss**. They are available in the standard ClusterWare yum repository and are compatible with non-Scyld RHEL and CentOS distributions - and perhaps with other distributions. One way to judge compatibility is to determine what libraries the ClusterWare daemons need to find on the non-Scyld compute node. (The daemons are known to execute in recent RHEL and CentOS environments.) Examine the daemons that were installed on the master node when ClusterWare was installed:

```
ldd /usr/sbin/sendstats
ldd /usr/sbin/kickbackproxy
```

and then determine if the libraries that these binaries employ are present on the target non-Scyld node. If the libraries do so exist, then the special ClusterWare packages can be downloaded and installed on a non-Scyld node.

First, you should download the packages from the ClusterWare yum repo. A useful downloader is the **/usr/bin/yumdownloader** utility, which can be installed from the CentOS *extras* yum repository if it is not already installed on your master node:

```
[root@cluster ~] # yum install yum-utils
```

Then use the utility to download the special Penguin ClusterWare rpms:

```
[root@cluster ~] # yumdownloader --destdir=<localdir> beostat-sendstats beonss-kickbackclient
```

retrieves the rpms and stores them into the directory <localdir>, e.g., `/var/www/html` or `/etc/beowulf/nonscyld`.

These special packages can be installed manually on the non-Scyld node, or can be installed as part of the kickstart procedure (see the Section called *Provisioning a non-Scyld node*). Each package includes a `/etc/init.d/` script that must be edited by the cluster administrator. Examine `/etc/init.d/beostat-sendstats` and `/etc/init.d/beonss-kickbackclient`, which contain comments that instruct the administrator about how to configure each script. Additionally, the non-Scyld node's `/etc/nsswitch.conf` must be configured to invoke the *kickback* service for the databases that the administrator wishes to involve **beonss** and the master node. See the master node's `/etc/beowulf/nsswitch.conf` for a guide to which databases are supported, e.g., *hosts*, *passwd*, *shadow*, and *group*. Finally, on the non-Scyld node, enable the scripts to start at node startup:

```
[root@cluster ~] # chkconfig beostat-sendstats on
```

```
[root@cluster ~] # chkconfig beonss-kickbackclient on
```

# Chapter 8. Managing Multiple Master Nodes

ClusterWare supports up to four master nodes on the same private cluster network. Every master node of a given cluster typically references a common `/etc/beowulf/config` file, which means all master nodes share a common understanding of all compute nodes that are attached to the private cluster network. That is, each master node knows a given physical node (denoted by its MAC addresses) by a common IP address and hostname. The `config` file's *masterorder* directive configures which master node controls which compute nodes. Additionally, every master node should share a common understanding of userID (`/etc/passwd`) and groupID (`/etc/group`) values.

## Active-Passive Masters

In a simple **active-passive** configuration, all compute nodes are "owned" by one and only one master node at any one time, and the secondary master node (or nodes) comes into play only if and when the primary master fails. A compute node's self-reassignment of ownership is called "cold re-parenting", as it only occurs when a node reboots.

For example, for a cluster with two master nodes and 32 compute nodes, the `/etc/beowulf/config` file on each master node contains the entry:

```
masterorder 0-31 10.1.1.1 10.1.1.2
```

or alternatively, an entry that uses a hyphen to avoid using explicit node numbers:

```
masterorder - 10.1.1.1 10.1.1.2
```

where the IP addresses are the static addresses assigned to the two masters. When a compute node boots, each master node interprets the same *masterorder* directive and knows that master 10.1.1.1 is the primary master and nominally "owns" all the nodes, and 10.1.1.2 is the secondary master which only steps in if the primary master is unresponsive.

## Active-Active Masters

Many labs and workgroups today have several compute clusters, where each one is dedicated to a different research team or engineering group, or is used to run different applications. When an unusually large job needs to execute, it may be useful to combine most or all of the nodes into a single larger cluster, and then afterwards split up the cluster when the job is completed. Also, the overall demand for particular applications may change over time, requiring changes in the allocation of nodes to applications.

The downside to this approach of using multiple discrete clusters, each with their separate private cluster network, is that the compute node reconfiguration requires physically rewiring the network cabling, or requires reprogramming a smart switch to move nodes from one discrete network to another.

However, with an **active-active** configuration, the cluster's master nodes and compute nodes reside on the same common private cluster network. The nodes are divided into subsets, and each subset is actively "owned" by a different master node and perhaps dedicated to separate users and applications. Additionally, each subset is passively associated with other master nodes.

For example, suppose each master node's `/etc/beowulf/config` contains:

```
masterorder  0-15 10.1.1.1 10.1.1.2 10.1.1.3
masterorder 16-31 10.1.1.2 10.1.1.1 10.1.1.3
```

which divides the 32 compute nodes into two subsets of 16, with one subset owned by master 10.1.1.1 and the other subset owned by 10.1.1.2. To add complexity to this example, we introduce a passive third master node, 10.1.1.3, which becomes active only if both master nodes fail. This configuration provides for several advantages over two discrete 16-node clusters.

One advantage is the same as provided by an **active-passive** configuration: in the event of a failure of one master node, that master's compute nodes automatically reboot and "cold re-parent" to another master node, which now becomes the active "owner" of all 32 compute nodes.

Another advantage is that the cluster administrator can easily respond to changing demands for computing resources through a controlled and methodical migration of nodes between masters. For example, the administrator can shift eight nodes, n16 to n23, from one master to the other by changing the *masterorder* entries to be:

```
masterorder  0-23 10.1.1.1 10.1.1.2 10.1.1.3
masterorder 24-31 10.1.1.2 10.1.1.1 10.1.1.3
```

and replicating this same change to all other master nodes. Then the administrator executes on every master node the command **service clusterware reload**, which instructs **beoserv** and **bpmaster** daemons to re-read the changed `/etc/beowulf/config`. Finally, on the currently "owning" master the administrator executes the command **bpctl -S 16-23 -R**, which reboots those shifted eight nodes and thereby causes them to cold re-parent to a different master node.

Reversing this reconfiguration, or performing any other reconfiguration, is equally simple:

1. Edit `/etc/beowulf/config` on one master to change the *masterorder* entries,
2. Replicate these same changes (or copy the same `config` file) to every affected master node,
3. Execute **service clusterware reload** on each master node to re-read the `config` file, and
4. Execute **bpctl -S** <*noderange*> **-R** on the current "owning" master node, where <*noderange*> is the range of affected nodes, which tells the affected node(s) to reboot and re-parent to their new active master.

# Chapter 9. Managing Node Failures

Node failures are an unfortunate reality of any computer system, and failures in a Scyld ClusterWare cluster are inevitable and hopefully rare. Various strategies and techniques are available to lessen the impact of node failures.

## Protecting an Application from Node Failure

There is only one good solution for protecting your application from node failure, and that is checkpointing. Checkpointing is where at regular intervals your application writes to disk what it has done so far, and at startup checks the file on disk so that it can start off where it was when it last wrote the file.

The way to checkpoint that gives you the highest chance of recovering is to send the data back to the master node and have it checkpoint there, and also make regular backups of your files on the master node.

When setting up checkpointing, it is important to think carefully about how often you want to checkpoint. Some jobs that don't have much data that needs to be saved can checkpoint as often as every 5 minutes, whereas if you have a large data set, it might be smarter to checkpoint every hour, day, week, or longer. It depends a lot on your application. If you have a lot of data to checkpoint, you don't want to do it often as that will drastically increase your run time. However, you also want to make sure that if you only checkpoint once every two days, that you can live with losing two days worth of work if there is ever a problem.

## Compute Node Failure

A compute node can fail for any of a variety of reasons, e.g., broken node hardware, a broken network, software bugs, or inadequate hardware resources. A common example of the latter is a condition known as *Out Of Memory*, or *OOM*, which occurs when one or more applications on the node have consumed all available RAM memory and no swap space is available. The Linux kernel detects an OOM condition, attempts to report what is happening to the cluster's syslog server, and begins to kill processes on the node in an attempt to eliminate the process that is triggering the problem. While this kernel response may occasionally be successful, more commonly it will kill one or more processes that are important for proper node behavior (e.g., a job manager daemon, the crucial Scyld **bpslave** daemon, or even a daemon that is required for the kernel's syslog messages to get communicated to the cluster's syslog server). When that happens, the node may still remain *up* in a technical sense, but the node is useless and must be rebooted.

### When Compute Nodes Fail

When a compute node fails, all jobs running on that node will fail. If there was an MPI job running that was using that node, the entire job will fail on all the nodes on which the MPI program was running.

Even though the running jobs running on that node failed, jobs running on other nodes that weren't communicating with jobs on the failed node will continue to run without a problem.

If the problem with the node is easily fixed and you want to bring the node back into the cluster, then you can try to reboot it using **bpctl -S** *nodenumber* **-R**. If the compute node has failed in a more catastrophic way, then such a graceful reboot will not work, and you will need to powercycle or manually reset the hardware. When the node returns to the *up* state, new jobs can be spawned that will use it.

If you wish to switch out the node for a new physical machine, then you must replace the broken node's MAC addresses with the new machine's MAC addresses. When you boot the new machine, it either appears as a new cluster node that is appended to the end of the list of nodes (if the `config` file says *nodeassign append* and there is room for new nodes), or else the node's MAC addresses get written to the `/var/beowulf/unknown_addresses` file. Alternatively, manually edit the `config` to change the MAC addresses of the broken node to the MAC addresses of the new machine, followed by the

command **service clusterware reload**. Reboot this node, or use IPMI to powercycle it, and the new machine reboots in the correct node order.

### Compute Node Data

What happens to data on a compute node after the node goes down depends on how you have set up the file system on the node. If you are only using a RAMdisk on your compute nodes, then all data stored on your compute node will be lost when it goes down.

If you are using the harddrive on your compute nodes, there are a few more variables to take into account. If you have your cluster configured to run **mke2fs** on every compute node boot, then all data that was stored on **ext2** file systems on the compute nodes will be destroyed. If **mke2fs** does not execute, then **fsck** will try to recover the **ext2** file systems; however, there are no guarantees that the file system will be recoverable.

Note that even if **fsck** is able to recover the file system, there is a possibility that files you were writing to at the moment of node failure may be in a corrupt or unstable state.

## Master Node Failure

A master node can fail for the same reasons a compute node can fail, i.e., hardware faults or software faults. An Out-Of-Memory condition is more rare on a master node because the master node is typically configured with more physical RAM, more swap space, and is less commonly a participant in user application execution than is a compute node. However, in a Scyld ClusterWare cluster the master node plays an important role in the centralized management of the cluster, so the loss of a master node for any reason has more severe consequences than the loss of a single compute node. One common strategy for reducing the impact of a master node failure is to employ multiple master nodes in the cluster. See Chapter 8 for details.

Another moderating strategy is to enable *Run-to-Completion*. If the **bpslave** daemon that runs on each compute node detects that its master node has become unresponsive, then the compute node becomes an *orphan*. What happens next depends upon whether or not the compute nodes have been configured for *Run-to-Completion*.

### When Master Nodes Fail - Without Run-to-Completion

The default behavior of an orphaned **bpslave** is to initiate a reboot. All currently executing jobs on the compute node will therefore fail. The reboot generates a new DHCP request and a PXEboot. If multiple master nodes are available, then eventually one master node will respond. The compute node reconnects to this master - perhaps the same master that failed and has itself restarted, or perhaps a different master - and the compute node will be available to accept new jobs.

Currently, Scyld only offers *Cold Re-parenting* of a compute node, in which a compute node must perform a full reboot in order to "fail-over" and reconnect to a master. See Chapter 8 for details.

### When Master Nodes Fail - With Run-to-Completion

You can enable *Run-to-Completion* by enabling the ClusterWare script:

```
beochkconfig 85run2complete on
```

When enabled, if the compute node becomes orphaned because its **bpslave** daemon has lost contact with its master node's **bpmaster** daemon, then the compute node does *not* immediately reboot. Instead, the **bpslave** daemon keeps the node up and running as best it can without the cooperation of an active master node. In an ideal world, most or all jobs running on

that compute node will continue to execute until they complete or until they require some external resource that causes them to hang indefinitely.

Run-to-Completion enjoys greatest success when the private cluster network uses file server(s) that require no involvement of any compute node's active master node. In particular, this means not using the master node as an NFS server, and not using a file server that is accessed using IP-forwarding through the master node. Otherwise, an unresponsive master also means an unresponsive file server, and that circumstance is often fatal to a job. Keep in mind that the default `/etc/beowulf/fstab` uses *$MASTER* as the NFS server. You should edit `/etc/beowulf/fstab` to change *$MASTER* to the IP address of the dedicated (and hopefully long-lived) non-master NFS server.

Stopping or restarting the *ClusterWare* service, or just rebooting the compute nodes doing **bpctl -S all -R**, will not put the compute nodes into an orphan state. These actions instruct each compute node to perform an immediate graceful shutdown and to restart with a PXEboot request to its active master node. Similarly, rebooting the master node will also stop the service with a **service clusterware stop** as part of the master shutdown, and the compute nodes will immediately reboot and attempt to PXEboot before the master node has fully rebooted and thus ready to service the nodes. This will be a problem unless another master node is running on the private cluster network that will respond to the PXEboot request, or unless the nodes' BIOS have been configured to perpetually retry the PXEboot, or unless you explicitly force all the compute nodes to immediately become orphans prior to rebooting the master with **bpctl -S all -O**, thereby delaying the nodes' reboots until the master has time to reboot.

Once a compute node has become orphaned, it can only rejoin the cluster by rebooting, i.e., a so-called *Cold Re-parenting*. There are two modes that **bpslave** can employ:

1. No automatic reboot. The cluster administrator must reboot each orphaned node using IPMI or by manually power-cycling the server(s).
2. Reboot the node after being "effectively idle" for a span of *N* seconds. This is the default mode. The default *N* is 300 seconds, and the default "effectively idle" is cpu usage below 1% of one cpu's available cpu cycles.

Edit the `85run2complete` script to change the defaults. Alternatively, the **bpctl** can set (or reset) the run-to-completion modes and values. See **man bpctl**.

The term "effectively idle" means a condition wherein the cpu usage on the compute node is so small as to be interpreted as insignificant, e.g., attributed to various daemons such as **bpslave**, **sendstats**, and **pbs_mom**, which periodically awaken, check fruitlessly for pending work, and quickly go back to sleep. An orphaned node's **bpslave** periodically computes cpu usage across short time intervals. If the cpu usage is below a threshold percentage *P* (default 1%) of one cpu's total available cpu cycles, then the node is deemed "effectively idle" across that short time interval. If and when the "effectively idle" condition persists for the full *N* seconds time span (default 300 seconds), then the node reboots. If the cpu usage exceeds that threshold percentage during any one of those short time intervals, then the time-until-reboot is reset back to the full *N* seconds.

If the cluster uses TORQUE as a job manager, Run-to-Completion works best if TORQUE is configured for High Availability. Refer to PBS TORQUE documentation for details. RHEL7CentOS7 has deprecated the earlier Heartbeat software in preference to Corosync.

# Chapter 10. Compute Node Boot Options

One of the unique advantages of Scyld ClusterWare is the fast and flexible boot procedure for compute nodes. The Scyld **BeoBoot** system is a combination of unified booting and a carefully designed light-weight compute node environment. The **BeoBoot** system allows compute nodes to initialize with a very small boot image that may be stored on a wide range of boot media. This small boot image never has to change; however, Scyld ClusterWare's boot setup allows you to change the kernel the compute nodes run, the modules that are loaded, and every aspect of the application environment by changing a few files on the master node.

This chapter gives instructions for setting up different types of boot media for the compute nodes, changing various settings that control the boot process, and checking for boot error messages. A detailed description of the boot process is included in the ClusterWare technical description in Chapter 1.

## Compute Node Boot Media

There are several ways to boot a compute node with Scyld ClusterWare, as discussed in the following sections. The methods described are all interchangeable, and they work seamlessly with each other. Thus, you can have some of your compute nodes boot using one method and other nodes boot with a different method.

### PXE

PXE is a protocol that defines a standard way to netboot x86-based machines. In order for PXE to work, your compute nodes must have support for it in both the network adapters as well as the BIOS. The option to PXE boot must also be turned on in the BIOS. This is the preferred method of booting nodes in a Scyld cluster.

### Local Disk

You can configure a node to boot from its local harddrive. See Chapter 7 for details.

### Linux BIOS

Linux BIOS is a project to replace the BIOS of a machine with Linux. This greatly speeds up the boot process as most of the actual work done by the BIOS is designed to make things like DOS work, but which aren't really needed by Linux.

There has been work done by third parties so that it is a Scyld ClusterWare initial image that replaces the BIOS. This has the advantage that all you need for a compute node is a motherboard with ram, processor, built-in network adapter, and a power supply.

Linux BIOS is not supported by Penguin Computing, Inc., however you can see http://www.linuxbios.org/ for more information if you are interested.

### Flash Disk

Although not Scyld specific, using a flash disk is mentioned as it can increase cluster reliability. A flash disk is a solid state device using an Electrical Erasable PROM (EEPROM). The devices are seen by the BIOS as an IDE or SCSI harddrive, and support all normal drive operations, including running **beofdisk** and installing the initial boot image. This allows a node cluster configuration with no moving parts other than cooling fans, and is an alternative to using the Linux BIOS. These devices are faster and cheaper than harddrives, and are currently limited to 4 MB to 512 MB. But, for booting, less than 2 MB would be needed.

# Changing Boot Settings

## Adding Steps to the node_up Script

If you wish to add more steps to be executed during the **node_up** script, you can do it without actually editing the script. Instead, you create a script in the `/etc/beowulf/init.d/` directory. All scripts in this directory will be executed for each node that boots up. This script will be sourced by the **node_up** script when the specified node boots, therefore it must be written in standard sh. When your script is sourced, the variable $NODE will be set to the node number that is booting. See the Section called *Site-Local Startup Scripts* in Appendix A for more details.

## Per-Node Parameters

Starting with Scyld Series 30, support is provided for specifying kernel image and kernel command line parameters on a per-node basis in the cluster config file `/etc/beowulf/config`. This enables one set of nodes to boot with a particular **initrd** image, while another group boots with a different one.

The utility of this feature can be illustrated by the use of the `memtest86` memory testing utility. For example, if you had just expanded your cluster with 5 new nodes (nodes 16 through 20), and you wanted to test their memory before putting them into production, you could have them all boot into `memtest86` rather than the usual Scyld **initrd** with the following entry in `/etc/beowulf/config`:

```
kernelimage 16-20 /var/beowulf/boot/memtest86.bin
initrdimage 16-20 none
kernelcommandline 16-20 none
```

## Other Per-Node Config Options

The cluster config file `/etc/beowulf/config` provides per-node support for node state changes, which allows the use of other scripts or tools to control and manipulate the *wake*, *alert*, and *down* states of nodes in the cluster.

# Error Logs

There are a number of ways to check for errors that occur during the compute node boot process, as follows:

- During the compute node boot process, any error messages are sent to the console of the compute node and forwarded to the cluster's syslog server's `/var/log/messages` file by the node's **beoklogd** daemon. By default, the syslog server is the master node. See the *syslog_server*= option in the Section called *Compute node command-line options* in Chapter 3 for details about how to direct these compute node logging messages to an alternate server. Messages can be viewed by manually editing this file, or by running the standard Linux System Logs tool: Select **System Tools -> System Logs** from the desktop menu to open the System Logs window, then select the System Log from the list of logs in the left panel, then scroll near the end to see errors.
- During each node's boot, the **node_up** script writes node-specific output to a log file `/var/log/beowulf/node.<nodenumber>`, where `<nodenumber>` is the node number. If the compute node ends up in the *error* state, or if it remains in the *boot* state for an extended length of time, then you should examine this node log with an editor.

## Notes

1. http://www.linuxbios.org/

# Chapter 11. Disk Partitioning

Partitioning allows disk storage space to be broken up into segments that are then accessible by the operating system. This chapter discusses disk partitioning concepts, the default partitioning used by Scyld ClusterWare, and some useful partitioning scenarios.

Scyld ClusterWare creates a RAM disk on the compute node by default during the initial boot process. This RAM disk is used to hold the final boot image downloaded from the master node. If you have diskless nodes, then this chapter does not pertain to you.

## Disk Partitioning Concepts

Disk partitioning on a cluster is essentially no different than partitioning on any stand-alone computer, with a few exceptions.

On a stand-alone computer or server, the disk drive's file system(s) divide the storage available on the disk into different sections that are configured in ways and sizes to meet your particular needs. Each partition is a segment that can be accessed independently, like a separate disk drive. The partitions are configured and determined by the partition table contained on each disk.

Each partition table entry contains information about the locations on the disk where the partition starts and ends, the state of the partition (active or not), and the partition's type. Many partition types exist, such as Linux native, AIX, DOS, etc.. The cluster administrator can determine the appropriate partition types for his/her own system.

Disk partitioning on a cluster is very much determined by the cluster system hardware and the requirements of the application(s) that will be running on the cluster, for instance:

- Some applications are very process intensive but not very data intensive. In such instances, the cluster may best utilize a RAM disk in the default partitioning scheme. The speed of the RAM will provide better performance, and not having a harddrive will provide some cost savings.
- Some applications are very data intensive but not very process intensive. In these cases, a hard disk is either required (given the size of the data set the application is working with) and/or is a very inexpensive solution over purchasing an equivalent amount of memory.

The harddrive partitioning scheme is very dependent on the application needs, the other tools that will interface with the data, and the preferences of the end-user.

## Disk Partitioning with ClusterWare

This section briefly describes the disk partitioning process for the master node and compute nodes in a Scyld cluster.

### Master Node

On the master node of a Scyld cluster, the disk partitioning administration is identical to that on any stand-alone Linux server. As part of installing Red Hat Linux, you are requested to select how you would like to partition the master node's hard disk. After installation, the disk partitioning can be modified, checked, and utilized via traditional Linux tools such as **fdisk**, **sfdisk**, **cfdisk**, **mount**, etc.

### Compute Nodes

The compute nodes of a Scyld cluster are slightly different from a traditional, stand-alone Linux server. Each compute node hard disk needs to be formatted and partitioned to be useful to the applications running on the cluster. However, not too many people would enjoy partitioning 64 or more nodes manually.

To simplify this task, Scyld ClusterWare provides the **beofdisk** tool, which allows remote partitioning of the compute node hard disks. It is very similar in operation to **fdisk**, but allows many nodes to be partitioned at once. The use of **beofdisk** for compute node partitioning is covered in more detail in the section on Partitioning Scenarios later in this chapter.

## Default Partitioning

This section addresses the default partitioning schemes used by Scyld ClusterWare.

### Master Node

The default Scyld partition table allocates 4 partitions:

- /boot partition
- /home partition
- / partition
- Swap partition = 2 times physical memory

Most administrators will want to change this to meet the requirements of their particular cluster.

### Compute Nodes

The default partition table allocates three partitions for each compute node:

- BeoBoot partition = 2 MB
- Swap partition = half the compute node's physical memory or half the disk, whichever is smaller
- Single root partition = remainder of disk

For diskless operation, the default method of configuring the compute nodes at boot time is to run off a RAM disk. This "diskless" configuration is appropriate for many applications, but not all. Typical usage requires configuration and partitioning of the compute node hard disks, which is covered in the partitioning scenarios discussed in the following section.

## Partitioning Scenarios

This section discusses how to implement two of the most common partitioning scenarios in Scyld ClusterWare:

- Apply the default partitioning to all disks in the cluster
- Specify your own manual but homogeneous partitioning to all disks in the cluster

The Scyld **beofdisk** tool can read an existing partition table on a compute node. It sequentially queries compute nodes beginning with node 0. For each new type/position/geometry it finds, it looks for an existing partition table file in `/etc/beowulf/fdisk`. If no partition table is present, a new one is generated that uses the default scheme. For each device/drive geometry it finds, **beofdisk** creates a file in `/etc/beowulf/fdisk/`. These files can then be modified by hand. Whether modified or using the default options, the files can be written back to the harddrives.

> **Caution**
>
> If you attempt to boot a node with an unpartitioned harddrive that is specified in `/etc/beowulf/fstab` (or a node-specific `fstab.`*N* for node *N*), then that node boots to an *error* state unless the `fstab` entry includes the "nonfatal" option. See the Reference Guide or **man beowulf-fstab** for details.

## Applying the Default Partitioning

To apply the default disk partitioning scheme (as recommended by the Scyld **beofdisk** tool) to the compute nodes, following these steps:

1. Query all the harddrives on the compute nodes and write out partition table files for them that contain the suggested partitioning:

   ```
   [root@cluster ~] # beofdisk -d
    Creating a default partition table for hda:2495:255:63
    Creating a default partition table for hda:1222:255:63
   ```

2. Read the partition table files, and partition the harddrives on the compute nodes so that they match:

   ```
   [root@cluster ~] # beofdisk -w
   ```

3. To use the new partitions you created, modify the `/etc/beowulf/fstab` file to specify how the partitions on the compute node should be mounted. The contents of `/etc/beowulf/fstab` should be in the standard `fstab` format.

4. To format the disk(s) on reboot, change "mkfs never" to "mkfs always" in the cluster config file `/etc/beowulf/config`.

5. To try out the new partitioning, reboot the compute nodes with the following:

   ```
   [root@cluster ~] # bpctl -S all -R
   ```

> **Caution**
>
> To prevent disks from being reformatted on subsequent reboots, change "mkfs always" back to "mkfs never" in `/etc/beowulf/config` after the nodes have booted.

## Specifying Manual Partitioning

You can manually apply your own homogeneous partitioning scheme to the partition tables, instead of taking the suggested defaults. There are two methods for doing this:

- The recommended method involves running **fdisk** on the first node (node 0) of the cluster, and then on every *first* node that has a unique type of hard disk.
- The other method is to manually edit the partition table text file retrieved by the **beofdisk** query.

For example, assume that your cluster has 6 compute nodes, and that all disks have 255 heads and 63 sectors (this is the most common). Nodes 0, 1, and 5 have a single IDE hard disk with 2500 cylinders. Nodes 2, 3, and 4 have a first IDE disk with 2000 cylinders, and node 4 has a SCSI disk with 5000 cylinders. This cluster could be partitioned as follows:

1. Partition the disk on node 0:

   ```
   [root@cluster ~] # bpsh 0 fdisk /dev/hda
   ```

   Follow the steps through the standard **fdisk** method of partitioning the disk.

2. Manually partition the disk on node 2 with **fdisk**:

   ```
   [root@cluster ~] # bpsh 2 fdisk /dev/hda
   ```

   Again, follow the steps through the standard **fdisk** method of partitioning the disk.

3. Manually partition the SCSI disk on node 4 with **fdisk**:

   ```
   [root@cluster ~] # bpsh 4 fdisk /dev/sda
   ```

   Again, follow the steps through the standard **fdisk** method of partitioning the disk.

4. Next, query the compute nodes to get all the partition table files written for their harddrives by using the command **beofdisk -q** .

   At this point, the 3 partition tables will be translated into text descriptions, and 3 files will be put in the directory `/etc/beowulf/fdisk`. The file names will be `hda:2500:255:63`, `hda:2000:255:63`, and `sda:5000:255:63`. These file names represent the way the compute node harddrives are currently partitioned.

   You have the option to skip the **fdisk** command and just edit these files manually. The danger is that there are lots of rules about what combinations of values are allowed, so it is easy to make an invalid partition table. Most of these rules are explained as comments at the top of the file.

5. Now write out the partitioning scheme using the command **beofdisk -w**.

   When specifying unique partitioning for certain nodes, you must also specify a unique `fstab` for each node that has a unique partition table. To do this, create the file `/etc/beowulf/fstab.<`*`nodenumber`*`>`. If this file exists, the **node_up** script will use that as the `fstab` for the compute node; otherwise, it will default to `/etc/beowulf/fstab`. Each instance of `/etc/beowulf/fstab.<`*`nodenumber`*`>` should be in the same format as `/etc/beowulf/fstab`.

6. To format the disk(s) on reboot, change "mkfs never" to "mkfs always" in the cluster config file `/etc/beowulf/config`.

7. To try out the new partitioning, reboot the compute nodes with the following:

   ```
   [root@cluster ~] # bpctl -S all -R
   ```

> **Caution**
>
> To prevent disks from being reformatted on subsequent reboots, change the "mkfs always" back to "mkfs never" in `/etc/beowulf/config` after the nodes have booted.

# Chapter 12. File Systems

## File Systems on a Cluster

File systems on a cluster consist of two types of file systems, local file systems and network file systems. The file `/etc/fstab` describes the filesystems mounted on the master node, and the file `/etc/beowulf/fstab` describes the filesystems mounted on each compute node. You may also create node-specific `/etc/beowulf/fstab.`*N* files, where *N* is a node number.

### Local File Systems

Local file systems are the file systems that exist locally on each machine. In the Scyld ClusterWare setup, the master node has a local file system, typically ext3, and each node also has a local file system. The local file systems are used for storing data that is local to the machines.

### Network/Cluster File Systems

Network file systems are used so that files can be shared across the cluster and every node in the cluster can see the exact same set of files. The default network file system for Scyld ClusterWare is NFS. NFS allows the contents of a directory on the server (by default the master node) to be accessed by the clients (the compute nodes). The default Scyld ClusterWare setup has the `/home` directory exported through NFS so that all the user home directories can be accessed on the compute nodes. Additionally, various other directories are mounted by default, as specified by `/etc/beowulf/fstab` or by a node-specific `fstab.`*N*.

Note that root's home directory is not in `/home`, and thus cannot access its home directory on the compute nodes. This should not be a problem, as normal compute jobs should not be run as "root".

## NFS

NFS is the standard way to have files stored on one machine, yet be able to access them from other machines on the network as if they were stored locally.

### NFS on Clusters

NFS in clusters is typically used so that if all the nodes need the same file, or set of files, they can access the file(s) through NFS. This way, if one changes the file, every node sees the change, and there is only one copy of the file that needs to be backed up.

### Configuration of NFS

The Network File System (NFS) is what Scyld ClusterWare uses to allow users to access their home directories and other remote directories from compute nodes. (The *User's Guide* has a small discussion on good and bad ways to use NFS.) Two files control what directories are NFS mounted on the compute nodes. The first is `/etc/exports`. This tells the nfs daemon on the master node what directories it should allow to be mounted and who can access them. Scyld ClusterWare adds various commonly useful entries to `/etc/exports`. For example:

```
/home  @cluster(rw)
```

The */home* says that `/home` can be nfs mounted, and *@cluster(rw)* says who can mount it and what forms of access are allowed. *@cluster* is a netgroup. It uses one word to represent several machines. In this case, it represents all your compute nodes. *cluster* is a special netgroup that is setup by **beonss** that automatically maps to all of your compute nodes. This makes it easy to specify something can be mounted by your compute nodes. The *(rw)* part specifies what permissions the compute node has when it mounts `/home`. In this case, all user processes on the compute nodes have read-write access to `/home`. There are more options that can go here, and you can find them detailed in **man exports**.

The second file is `/etc/beowulf/fstab`. (Note that it is possible to set up an individual `fstab.`*N* for a node. For this discussion, we will assume that you are using a global `fstab` for all nodes.) For example, one line in the default `/etc/beowulf/fstab` is the following:

```
$MASTER:/home  /home  nfs  nolock,nonfatal  0 0
```

This is the line that tells the compute nodes to try to mount `/home` when they boot:

- The *$MASTER* is a variable that will automatically be expanded to the IP of the master node.
- The first */home* is the directory location on the master node.
- The second */home* is where it should be mounted on the compute node.
- The *nfs* specifies that this is an nfs file system.
- The *nolock* specifies that locking should be turned off with this nfs mount. We turn off locking so that we don't have to run daemons on the compute nodes. (If you need locking, see the Section called *File Locking Over NFS* for details.)
- The *nonfatal* tells ClusterWare's `/usr/lib/beoboot/bin/setup_fs` script to treat a mount failure as a nonfatal problem. Without this *nonfatal* option, any mount failure leaves the compute node in an *error* state, thus making it unavailable to users.
- The two 0's on the end are there to make the `fstab` like the standard `fstab` in `/etc`.

To add an nfs mount of `/foo` to all your compute nodes, first add the following line to the end of the `/etc/exports` file:

```
/foo  @cluster(rw)
```

Then execute **exportfs -a** as root. For the mount to take place the next time your compute nodes reboot, you must add the following line to the end of `/etc/beowulf/fstab`:

```
$MASTER:/foo  /foo  nfs  nolock  0 0
```

You can then reboot all your nodes to make the nfs mount happen. If you wish to mount the new exported filesystem without rebooting the compute nodes, you can issue the following two commands:

```
[root @cluster ~] # bpsh -a mkdir -p /foo
[root @cluster ~] # bpsh -a mount -t nfs -o nolock master:/foo /foo
```

Note that `/foo` will need to be adjusted for the directory you actually want.

If you wish to stop mounting a certain directory on the compute nodes, you can either remove the line from `/etc/beowulf/fstab` or just comment it out by inserting a '#' at the beginning of the line. You can leave untouched the entry referring to the filesystem in `/etc/exports`, or you can delete the reference, whichever you feel more comfortable with.

If you wish to unmount that directory on all the compute nodes without rebooting them, you can then run the following:

```
[root @cluster ~] # bpsh -a umount /foo
```

where `/foo` is the directory you no longer wish to have NFS mounted.

> **Caution**
>
> On compute nodes, NFS directories must be mounted using either a specific IP address or the *$MASTER* keyword; the hostname cannot be used. This is because `fstab` is evaluated before node name resolution is available.

### File Locking Over NFS

By default, the compute nodes mount NFSv3 filesystems with locking turned off. If you have a program that requires locking, edit `/etc/beowulf/fstab` to remove the *nolock* keyword from the NFS mount entries.

Finally, reboot the cluster nodes to effect the NFS remounting with locking enabled.

### NFSD Configuration

By default, when the master node reboots, the `/etc/init.d/nfs` script launches 8 NFS daemon threads to service client NFS requests. For large clusters this count may be insufficient. One symptom of an insufficiency is a syslog message, most commonly seen when you boot all the cluster nodes:

```
nfsd: too many open TCP sockets, consider increasing the number of nfsd threads
```

To increase the thread count (e.g., to 16):

```
[root @cluster ~] # echo 16 > /proc/fs/nfsd/threads
```

Ideally, the chosen thread count should be sufficient to eliminate the syslog complaints, but not significantly higher, as that would unnecessarily consume system resources. To make the new value persistent across master node reboots, create the file `/etc/sysconfig/nfs`, if it does not already exist, and add to it an entry of the form:

```
RPCNFSDCOUNT=16
```

A value of 1.5x to 2x the number of nodes is probably adequate, although perhaps excessive.

A more refined analysis starts with examining NFSD statistics:

```
[root @cluster ~] # grep th /proc/net/rpc/nfsd
```

which outputs thread statistics of the form:

```
th 16 10 26.774 5.801 0.035 0.000 0.019 0.008 0.003 0.011 0.000 0.040
```

From left to right, the *16* is the current number of NFSD threads, and the *10* is the number of times that all threads have been simultaneously busy. (Not all circumstances of all threads being busy results in that syslog message, but a high all-busy count does suggest that adding more threads may be beneficial.)

The remaining 10 numbers are histogram buckets that show how many accumulated seconds a percentage of the total number of threads have been simultaneously busy. In this example, 0-10% of the threads were busy *26.744* seconds, 10-20% of the threads were busy *5.801* seconds, and 90-100% of the threads were busy *0.040* seconds. High numbers at the end indicate that most or all of the threads are simultaneously busy for significant periods of time, which suggests that adding more threads may be beneficial.

# ROMIO

ROMIO is a high-performance, portable implementation of MPI-IO, the I/O chapter in MPI-2: Extensions to the Message Passing Interface, and is included in the Scyld ClusterWare distribution. ROMIO is optimized for noncontiguous access patterns, which are common in parallel applications. It has an optimized implementation of collective I/O, an important optimization in parallel I/O.

## Reasons to Use ROMIO

ROMIO gives you an abstraction layer on top of high performance input/output. The details for the file system may be implemented in various ways, but ROMIO prevents you from caring. Your binary code will run on an NFS file system here and a different file system there, without changing a line or recompiling. Although POSIX open(), read(), ... calls already do this, the virtual file system code to handle this abstraction is deep in the kernel.

You may need to use ROMIO to take advantage of new special and experimental file systems. It is easier and more portable to implement a ROMIO module for a new file system than a Linux-specific VFS kernel layer.

Since ROMIO is an abstraction layer, it has the freedom to be implemented arbitrarily. For example, it could be implemented on top of the POSIX Asynchronous and List I/O calls for real-time performance reasons. The end-user application is shielded from caring, and benefits from careful optimization of the I/O details by experts.

## Installation and Configuration of ROMIO

### ROMIO Over NFS

To use ROMIO on NFS, file locking with **fcntl** must work correctly on the NFS installation. First, since file locking is turned off by default, you need to turn on NFSv3 locking. See the Section called *File Locking Over NFS*. Now, to get the **fcntl** locks to work, you must mount the NFS file system with the *noac* option (no attribute caching). This is done by modifying the line for mounting `/home` in `/etc/beowulf/fstab` to look like the following:

```
$MASTER:/home  /home  nfs  noac,nonfatal  0 0
```

Turning off attribute caching may reduce performance, but it is necessary for correct behavior.

# Other Cluster File Systems

There are variety of network file systems that can be used on a cluster. If you have questions regarding the use of any particular cluster file system with Scyld ClusterWare, contact Scyld Customer Support for assistance.

# Chapter 13. Load Balancing

You have made some rather significant investment in your cluster. It is also evident that it depreciates at a rather frightening rate. Given these two facts it should be obvious you want your cluster busy 100% of the time if possible.

However, timely results of output are also important. If the memory requirements of programs running on the cluster exceed the available physical memory, swap memory (hard disk) will be used severely reducing performance. Even if the memory requirements of many processes still fit within the physical memory, results of any one of the programs may take significantly longer to achieve if many jobs are running on the same nodes simultaneously.

Thus we come to concept of the "load balancing", which maintain a delicate balance between overburdened and idle. Load balancing is when multiple servers can perform the same task, and which server performs the task is based on which server is currently doing the least amount of work. This helps to spread a heavy work load across several machines, and does it intelligently; if one machine is more heavily loaded than the others, new requests will not be sent to it. By doing this, a job is always run on a machine that has the most resources to devote to it, and therefore gets finished sooner.

Generally, it is believed that a constant load of one 100% CPU bound process per CPU is ideal. However, not all processes are CPU bound; many are I/O bound on either the harddrive or the network. The act of load balancing is often described as "scheduling".

Optimal load balancing is almost never achieved; hence, it is a subject of study for many researchers. The optimal algorithm for scheduling the programs running on your cluster is probably not the same as it might be for others, so you may want to spend time on your own load balancing scheme.

## Load Balancing in a Scyld Cluster

Scyld ClusterWare supplies a general load balancing and job scheduling scheme via the **beomap** subsystem in conjunction with job queuing utilities. Mapping is the assignment of processes to nodes based on current CPU load. Queuing is the holding of jobs until the cluster is idle enough to let the jobs run. Both of these are covered in detail in other sections of this guide and in the *User's Guide*. In this section, we'll just discuss the scheduling policy that is used.

### Mapping Policy

The current default mapping policy consists of the following steps:

- Run on nodes that are idle
- Run on CPUs that are idle
- Minimize the load per CPU

Each proceeding step is only performed if the number of desired processes (NP) is not yet satisfied. The information required to perform these steps comes from the **BeoStat** sub-system of daemons and libbeostat library.

### Queuing Policy

The current default queuing policy is to attempt to determine the desired number of processes (NP) and other mapping parameters from the job script. Next, the **beomap** command is run to determine which nodes would be used if it ran immediately. If every node in the returned map is below 0.8 CPU usage the job is released for execution.

## Implementing a Scheduling Policy

The queuing portion of the schedule policy depends on which scheduling and resource management tool you are using. The mapping portions, however, are already modularized. There are a number of ways to override the default, including

- Substitute a different program for the **beomap** command and use **mpirun** to start jobs (which uses beomap).
- Create a shared library that defines the function get_beowulf_job_map() and use the environment variable LD_PRELOAD to force the pre-loading of this shared library.
- Create the shared library and replace the default `/usr/liblibbeomap.so` file.

These methods are in order of complexity. We can't actually highly recommend the first method as your mileage may vary. The second method is the most recommended followed by the third method of replacing the Scyld source code when you're happy that your scheduler is better.

It is highly recommended that you get the source code for the **beomap** package. It will give you a head start on writing your own mappers. For more information on developing your own mapper, see the *Programmer's Guide*.

# Chapter 14. IPMI

Included in the Scyld ClusterWare distribution are extra tools that may be of interest to users, including **IPMITool** for monitoring and managing compute node hardware.

## IPMITool

IPMITool is a hardware management utility that supports the Intelligent Platform Management Interface (IPMI) specification v1.5 and v2.0.

IPMI is an open standard that defines the structures and interfaces used for remote monitoring and management of a computer motherboard (baseboard). IPMI defines a micro-controller, called the "baseboard management controller" (BMC), which is accessed locally through the managed computer's bus or through an out-of-band network interface connection (NIC).

For ClusterWare, Scyld supports IPMITool as the primary way to monitor and manage compute node hardware. IPMITool is part of the OpenIPMI package included in the base distribution (RHEL4 Update 2 and later). The Scyld ClusterWare distribution includes `/etc/beowulf/init.d/20ipmi`, a script that enables IPMI on compute nodes.

You can use IPMITool for a variety of tasks, such as:

- Inventory your computer baseboards to determine what sensors are present
- Monitor sensors (fan status, temperature, power supply voltages, etc.)
- Read and display values from the Sensor Data Repository (SDR)
- Read and set the BMC's LAN configuration
- Remotely control chassis power
- Display the contents of the System Event Log (SEL), which records events detected by the BMC as well as events explicitly logged by the operating system
- Print out Field Replaceable Unit (FRU) information, such as vendor ID, manufacturer, etc.
- Configure and emulate a serial port to the baseboard using the out-of-band network connection known as serial over LAN (SOL)

Several dozen companies support IPMI, including many leading manufacturers of computer hardware. You can learn more about OpenIPMI from the OpenIPMI project page at http://openipmi.sourceforge.net, which includes links to documentation and downloads.

## Notes

1. http://openipmi.sourceforge.net

# Chapter 15. Updating Software On Your Cluster

From time to time, Scyld may release updates and add-ons to Scyld ClusterWare. Customers on active support plans for Scyld software products can access these updates on the Penguin Computing website. Visit http://www.penguincomputing.com/support for details. This site offers answers to common technical questions and provides access to application notes, software updates, product documentation, and *Release Notes*.

The *Release Notes* for each software update will include instructions for installation, along with information on why the update was released and what bug(s) it fixes. Be sure to thoroughly read the *Release Notes*, as they may discuss specific requirements and potential conflicts with other software.

## What Can't Be Updated

Some packages provided with Scyld ClusterWare, such as Ganglia, are specifically optimized to take advantage of the **BProc** unified process space, which is added to the standard Linux distributions that Scyld supports. Other packages, such as MPICH2, MVAPICH2, MPICH3, and OpenMPI, take advantage of features of the Scyld ClusterWare TORQUE distribution. Although there are generally available versions of these packages that you can download from other sources, you should use the versions provided by Scyld for best performance with BProc and ClusterWare. Contact Scyld Customer Support if you have questions about specific packages that you would like to use with Scyld ClusterWare.

Users may also choose to use commercially available MPIs, such as Intel, HP, Scali, or Verari. These require specific configuration on Scyld ClusterWare. See the Penguin Computing Support Portal at http://www.penguincomputing.com/support, or contact Scyld Customer Support.

## Notes

1. http://www.penguincomputing.com/support
2. http://www.penguincomputing.com/support

# Appendix A. Special Directories, Configuration Files, and Scripts

Scyld ClusterWare adds some special files and directories on top of the standard Linux install that help control the behavior of the cluster. This appendix contains a summary of those files and directories, and what is in them.

## What Resides on the Master Node

### /etc/beowulf/ directory

All the config files for controlling how **BProc** and **Beoboot** behave are stored here.

#### /etc/beowulf/config

This file contains the settings that control the **bpmaster** daemon for **BProc**, and the **beoserv** daemon that is part of beoboot. It also contains part of the configuration for how to make beoboot boot images.

#### /etc/beowulf/fdisk/

This directory is used by **beofdisk** to store files detailing the partitioning of the compute nodes' harddrives, and is also read from when it rewrites the partition tables on the compute nodes. See Chapter 11 for more information on disk partitioning.

#### /etc/beowulf/fstab

Refer to Chapter 11 for details on using node-specific `fstab.`*N* files.

#### /etc/beowulf/backups/ directory

Contains time-stamped backups of older versions of various configuration files, e.g., `/etc/beowulf/config` and `/etc/beowulf/fstab`, to assist in the recovery of a working configuration after an invalid edit.

#### /etc/beowulf/init.d/ directory

Contains various scripts that are executed on the master node by the **node_up** script when booting a compute node.

#### /etc/beowulf/conf.d/ directory

Contains various configuration files that are needed when booting a compute node.

### /usr/lib/beoboot directory

This directory contains files that are used by beoboot for booting compute nodes.

### /usr/lib/beoboot/bin

This directory contains the **node_up** script and several smaller scripts that it calls.

## /var/beowulf directory

This directory contains compute node boot files and static information, as well as the list of unknown MAC addresses. It includes three subdirectories.

### /var/beowulf/boot

This is the default location for files essential to booting compute nodes. Once a system is up and running, you will typically find three files in this directory:

- `computenode` — the boot sector used for bootstrapping the kernel on the compute node.
- `computenode.initrd` — the kernel image and initial ramdisk used to boot the compute node.
- `computenode.rootfs` — the root file system for the compute node.

### /var/beowulf/statistics

This directory contains a cached copy of static information from the compute nodes. At a minimum, it includes a copy of `/proc/cpuinfo`.

### /var/beowulf/unknown_addresses

This file contains a list of Ethernet hardware (MAC) addresses for nodes considered *unknown* by the cluster. See the Section called *Compute Node Categories* in Chapter 1 for more information.

## /var/log/beowulf directory

This directory contains the boot logs from compute nodes. These logs are the output of what happens when the **node_up** script runs. The files are named `node.<number>`, where <*number*> is the actual node number.

# What Gets Put on the Compute Nodes at Boot Time

- Generally speaking, the `/dev` directory contains a subset of devices present in the `/dev` directory on the master node. The `/usr/lib/beoboot/bin/mknoderootfs` script creates most of the `/dev/` entries (e.g., `zero`, `null`, and `random`). `/etc/beowulf/init.d/20ipmi` creates `ipmi0`. `/usr/lib/beoboot/bin/setup_fs` creates `shm` and `pts` (as directed by `/etc/beowulf/fstab`). The harddrive devices (e.g., `sda`) are created at compute node bootup time, if local drives are discovered. If Infiniband hardware is present on the compute node, `/etc/beowulf/init.d/15openib` creates various device entries in `/dev/infiniband/`.

- The `/etc` directory contains the `ld.so.cache`, `localtime`, `mtab`, and `nsswitch.conf` files. The **node_up** script creates a simple `hosts` file.
- The `/home` directory exists as a read-write NFS mount of the `/home` directory from the master node. Thus, all the home directories can be accessed by jobs running on the compute nodes.
- Additionally, other read-only NFS mounts exist by default, to better assist out-of-the-box application and script execution: `/bin`, `/usr/bin`, `/opt`, `/usr/lib64/python2.3`, `/usr/lib/perl5`, and `/usr/lib64/perl5`.
- The **node_up** script mounts pseudo-filesystems as directed by `/etc/beowulf/fstab`: `/proc`, `/sys`, and `/bpfs`.
- `mknoderootfs` creates `/var` and several of its subdirectories.
- The `/tmp` directory is world-writeable and can be used as temporary space for compute jobs.
- `/etc/beowulf/config` names various *libraries* directories that are managed by the compute node's library cache. Run **beoconfig libraries** to see the current list of library directories. Caching shared libraries, done automatically as needed on a compute node, speeds up the transfer process when you are trying to run jobs, eliminates the need to NFS-mount the various common directories that contain libraries, and minimizes the space consumed by libraries in the compute node's RAM filesystem.
- Typically, when the loader starts up an application, it opens the needed shared libraries. Each open() causes the compute node to pull the shared library from the master node and save it in the library cache, which typically resides in the node's RAM filesystem. However, some applications and scripts reference a shared library or other file that, although it resides in one of those *libraries* directories, the reference does not use open() to access the file, and so the file does not get automatically pulled into the library cache. For example, an application or script might first use stat() to determine if a specific file exists, and then use open() if the stat() is successful, otherwise continue on to stat() an alternative file. The stat() on the compute node will fail until an open() pulls the file from the master. The application or script thus fails to execute, and the missing library or file name is typically displayed as an error.

  To remedy this type of failure, you should use a *prestage* directive in `/etc/beowulf/config` to explicitly name files that should be pulled to each compute node at node startup time. Run **beoconfig prestage** for the current list of prestaged files.

## /usr/lib/locale/locale-archive Internationalization

Glibc applications silently open the file `/usr/lib/locale/locale-archive`, which means it gets downloaded by each compute node early in a node's startup sequence via the BProc filecache functionality. The default `locale-archive` is 54 MBytes in RHEL5 and 95 MBytes in RHEL6. This download consumes significant network bandwidth and thus causes serialization delays if numerous compute nodes attempt to concurrently boot, and thereafter this large file consumes significant RAM filesystem space on each node. It is likely that a cluster's users and applications do not require all the international locale data that is present in the default file. With care, the cluster administrator may choose to rebuild `locale-archive` with a greatly reduced set of locales and thus create a significantly smaller file that is less impactful on cluster performance.

Rebuilding and replacing `locale-archive` should be done on a quiescent master node, as the file typically is mmap'ed by a process (e.g., **crond**, **bash**), and the appearance of a replacement version may perturb shells and other programs, such as aborting the shell that executes the rebuild or having that shell issue an immediate warning message about an undefined environment variable. In the event that a problem does appear, you should reboot the master node. Otherwise, newly executing programs on the master node will use the updated `locale-archive`, and compute nodes will employ the new file only after the node reboots.

In a **RHEL5** environment, the *glibc-common* RPM installs the `/usr/lib/locale/` directory containing the full set of locale definition files and a full `locale-archive` binary file. The **build-locale-archive** command rebuilds the `locale-archive` with every individual locale data file that is found in that directory. Thus, to reduce the size of `locale-archive`, you must first reduce the number of locale data files in that directory - but only after saving the default

locale data files in a safe place, so you can later rebuild the `locale-archive` with a different set of locale data files as the cluster's needs change. Beginning with the default `/usr/lib/locale/` directory with its full set of locale data files:

```
[root@cluster ~] # cd /usr/lib
[root@cluster ~] # cp -a locale locale.default
[root@cluster ~] # (cd locale ; rm -fr *_*)
```

saves all the locale data files in a new directory and produces a stripped-down `/usr/lib/locale/`, leaving only the `locale-archive` file. Now reintroduce a smaller set of locale data files. For example, to include the U.S.-English and U.S.-Great Britain locale files:

```
[root@cluster ~] # cp -a locale.default/en_US* locale
[root@cluster ~] # cp -a locale.default/en_GB* locale
```

When `/usr/lib/locale/` contains the desired locale data files, perform the rebuild:

```
[root@cluster ~] # /usr/sbin/build-locale-archive
```

and reboot the master node and/or the compute nodes as needed.

In a **RHEL6** environment, the *glibc-common* RPM installs just the full default `locale-archive` binary file. The default `/usr/lib/locale/` directory contains no locale data files. Scyld ClusterWare has saved the default `locale-archive` as `locale-archive.default` and has created `locale-archive.default.list` as a text file containing a list of all the locales in that default file. To generate a smaller file, you start with the full default `locale-archive`, then eliminate locales from the full list using **localedef --delete-from-archive**, then execute **build-locale-archive** to finalize the new `locale-archive` file. To assist in this procedure, Scyld ClusterWare installs helper scripts and some sample locale lists. For example, to rebuild with just the U.S.-English locales:

```
[root@cluster ~] # cd /usr/lib/locale
[root@cluster ~] # ./rebuild-archive.sh locales.English_US
```

Or to include all the English language locales:

```
[root@cluster ~] # cd /usr/lib/locale
[root@cluster ~] # ./rebuild-archive.sh locales.English
```

When executing **rebuild-archive.sh**, this helper script prints details of what is being requested and asks for permission to proceed.

Several other sample `locales.*` files have been provided. The local cluster administrator can use one of these files, or can create a new custom file, as desired. Each such `locales.*` file should contain a list of one or more specific locales (e.g., *en_US.uts8*), or contain patterns that match a locale or locales (e.g., *en_US*), one per line. For example, the `locales.English` file contains:

```
# All English language locales
en_
```

which is a pattern that matches every en_* locale.

Additionally, Scyld ClusterWare provides **reset-archive.sh**, which is a script that returns `locale-archive` to its original default state.

> **Caution**
>
> Note that for both RHEL5 and RHEL6, we recommend always including *en_US** locales, just to be safe, as the default RHEL/CentOS distributions reference the `LANG=en_US.uts8` locale in several `/etc/` configuration files. Each Scyld ClusterWare6-supplied `locales.*` file contains the suggested *en_US* locale pattern.

## Site-Local Startup Scripts

Local, homegrown scripts to be executed at node boot time can be placed in `/etc/beowulf/init.d/`. The conventions for this are as follows:

- Scripts should live in `/etc/beowulf/init.d/`
- Scripts should be numbered in the order in which they are to be executed (e.g., 20raid, 30startsan, 45mycustom_hw)
- Any scripts going into `/etc/beowulf/init.d/` should be cluster aware. That is, they should contain the appropriate **bpsh** and/or **bpcp** commands to make the script work on the compute node rather than on the master node. Examine the Scyld ClusterWare-distributed scripts for examples.

Any local modifications to Scyld ClusterWare-distributed scripts in `/etc/beowulf/init.d` will be lost across subsequent Scyld ClusterWare updates. If a local sysadmin believes a local modification is necessary, we suggest:

1. Copy the to-be-edited original script to a file with a unique name, e.g.:

   ```
   cd /etc/beowulf/init.d
   cp 37some_script 37some_script_local
   ```

2. Remove the executable state of the original:

   ```
   beochkconfig 37some_script off
   ```

3. Edit `37some_script_local` as desired.
4. Thereafter, subsequent ClusterWare updates may install a new `37some_script`, but the update will not re-enable the non-executable state of that script. The local `37some_script_local` remains untouched. However, keep in mind that the newer ClusterWare version of `37some_script` may contain fixes or other changes that need to be reflected in `37some_script_local` because that edited file was based upon an older ClusterWare version.

## Sample Kickstart Script

Non-Scyld nodes can be provisioned using the Red Hat **kickstart** utility. The following is a sample kickstart configuration script, which should be edited as appropriate for your local cluster:

```
# centos 5u3  (amd64) hybrid example kickstart

install
reboot
# point to NFS server that exports a directory containing the iso images of centOS 5.3
nfs --server=192.168.5.30 --dir=/eng_local/nfs-install/centos5u3_amd64
lang en_US.UTF-8
keyboard us
xconfig --startxonboot
network --device eth0 --bootproto dhcp --onboot yes
```

```
#network --device eth1 --onboot no --bootproto dhcp
rootpw --iscrypted $1$DC2r9BD4$Y1QsTSuL6K9ESdVk18eJT0
firewall --disabled
selinux --disabled
authconfig --enableshadow --enablemd5
timezone --utc America/Los_Angeles
bootloader --location=mbr
key --skip

# The following is commented-out so nobody uses this by accident and
# overwrites their local harddisks on a compute node.
#
# In order to enable using this kickstart script to install an operating system
# on /dev/sda of your compute node and thereby erasing all prior content,
# remove the comment character in front of the next 4 lines:

# clearpart --linux --drives=sda
# part /boot --fstype ext3 --size=100 --ondisk=sda
# part swap --fstype swap --size=2040 --ondisk=sda
# part / --fstype ext3 --size=1024 --grow

##############################################################################
%packages
@ ruby
@ system-tools
@ MySQL Database
@ Editors
@ System Tools
@ Text-based Internet
@ Legacy Network Server
@ DNS Name Server
@ FTP Server
@ Network Servers
@ Web Server
@ Server Configuration Tools
@ Sound and Video
@ Administration Tools
@ Graphical Internet
@ Engineering and Scientific
@ Development Libraries
@ GNOME Software Development
@ X Software Development
@ Authoring and Publishing
@ Legacy Software Development
@ Emacs
@ Legacy Software Support
@ Ruby
@ KDE Software Development
#@ Horde
@ PostgreSQL Database
@ Development Tools
#@ Yum Utilities
#@ FreeNX and NX
kernel-devel
OpenIPMI-tools
```

```
openmpi-devel
sg3_utils

###############################################################################
%pre

# any thing you want to happen before the install process starts

###############################################################################
%post
#!/bin/bash
# anything you want to happen after the install process finishes

masterip=10.56.10.1
wget http://$masterip/sendstats
chmod +x sendstats
mv sendstats /usr/local/sbin/
echo "/usr/local/sbin/sendstats" >> /etc/rc.local

# If you get the blinking cursor of death and no OS post, then uncomment this.
#grub-install --root-directory=/boot hd0
#grub-install --root-directory=/boot hd1
#grub-install --root-directory=/boot hd2

# Removes rhgb and quiet from grub.conf
sed -i /boot/grub/grub.conf -e 's/rhgb//g;s/quiet//g'

# Sets up the serial console in grub.conf
# TODO

# changes xorg.conf from mga to vesa
sed -i /etc/X11/xorg.conf -e 's/mga/vesa/'

# turns on ipmi
systemctl enable ipmievd
systemctl enable sshd
wget http://10.56.10.1/done
```