NPACI Rocks Cluster Distribution: Reference Guide

Reference Guide for NPACI Rocks version 4.1 Edition

NPACI Rocks Cluster Distribution: Reference Guide :
Reference Guide for NPACI Rocks version 4.1 Edition
Published 2004

Table of Contents

List of Tables

Chapter 1. Introduction

This book gives a Reference Guide for the Rocks Cluster Distribution. We intend for this publication to illuminate the specifics of our system. Intead of giving philosophies or behavior, this guide is a hard-core software and API reference. Essentially we provide a blueprint and a lesson on the languages used in the Rocks Cluster Distribution.

Included in this volume are chapters on Roll development, Kickstart XML, Distribtions, and Monitoring APIs. We indend this book for the Rocks developer, who is anyone who will add to the Rocks system. In each section of this document, we strive to give the developer enough information to expand and extend our structures.

Chapter 2. Roll Developers Guide

Rocks uses Rolls to install software components. This Chapter tells you how to build your own Roll.

2.1. Roll Files

2.1.1. Roll Development

This section guides developers on how to build a Roll.

To get started, login to a Rocks frontend and mirror the Rocks distribution[1] and perform a CVS checkout of the Rocks code[2].

After the CVS checkout is completed, go to the top-level roll directory:

```
# cd rocks/src/roll
```

You should see something similar to:

```
bin           cacl    etc            hpc       metaroll.mk  patch
birn          condor  gfarm          intel     nbcr         pbs
birn-oracle1  CVS     grid           java      ninf         sge
birn-oracle2  demo    grid-rocks-ca  Makefile  optiputer    top500
```

Now create a new directory for your roll -- the direcory name will be the name of your roll. For example, let's say you want your roll to be called *skel*:

```
# mkdir skel
```

To describe what directories and files you need to build your roll, we'll use the `intel` roll as a guide. In the directory `intel`, you'll see:

```
CVS  graphs  Makefile  nodes  RPMS  src  SRPMS  version.mk
```

Copy the files `Makefile` and `version.mk` into your roll:

```
# cp Makefile version.mk ../skel/
```

The first file, `Makefile`, should remain unchanged. It includes other files in order to drive the roll building process.

The second file, `version.mk`, should only be modified if you wish to change the version number or release number of your roll.

Now make the directories: `graphs, nodes, RPMS, SRPMS and src`. Your roll should look like:

```
< roll name > ----+
           |
           +- graphs -+
           |
           +- nodes -+
           |
```

```
+- RPMS -+
|
+- SRPMS -+
|
+- src -+
|
+- version.mk
|
+- Makefile
```

Inside the `nodes` directory, you will put your Rocks-based XML configuration files. These files are used to install packages and to configure their respective services.

In the Intel Roll, the file `roll/intel/nodes/intel.xml` has the package names for the compilers to be installed on specific hardware architectures. For example, the `intel-icc8` is installed with the line:

```
<package>intel-icc8</package>
```

2.1.2. Nodes Directory

Inside the `nodes` directory, you will put your Rocks-based XML configuration files. These files are used to install packages and to configure their respective services. See the Kickstart XML section for node file syntax.

In the Intel Roll, the file `roll/intel/nodes/intel.xml` has the package names for the compilers to be installed on specific hardware architectures. For example, the `intel-icc8` is installed with the line:

```
<package>intel-icc8</package>
```

The `<post>` section of the node `roll/intel/nodes/intel.xml` shows how the packages are configured after they are installed.

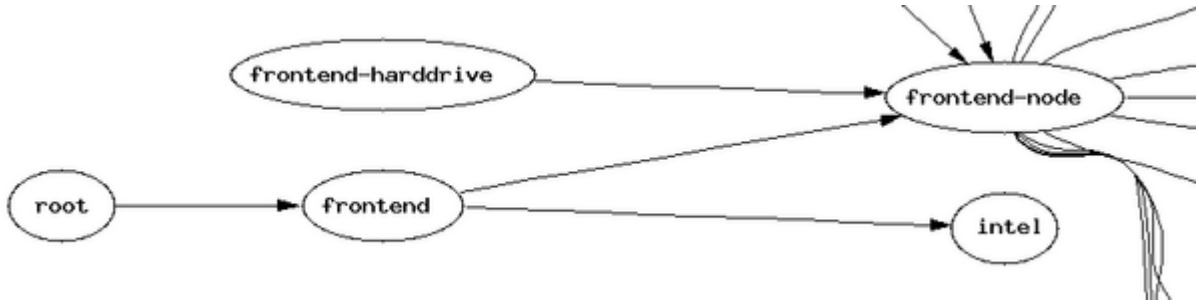
2.1.3. Graphs Directory

Inside the `graphs` directory, you should create another directory named `default`.

Looking back at the Intel Roll, there is one file in its `graphs/default` directory called: `roll/intel/graphs/default/intel.xml`.

This file describes how all the files in the `nodes` directory are linked together in the kickstart graph. For example, in `roll/intel/graphs/default/intel.xml` we see that the XML configuration node file `roll/intel/nodes/intel.xml` is linked into the `frontend` node.

To see how this works, below is an excerpt from the kickstart graph when the Intel Roll is installed:



The edge which travels from the node labeled *frontend* to the node labeled *intel* is result of the specification found in `roll/intel/graphs/default/intel.xml` which looks like:

```
<edge from="frontend">
        <to>intel</to>
</edge>
```

See the Graph XML section for graph file syntax.

2.1.3.1. Using a Graph File to Add Packages

This section describes how to use the nodes and graphs files to direct the installer to add packages to the frontend or the compute nodes.

Say you have 4 packages and they're named: package-one, package-two, package-three and package-four. You want to add package-one and package-two to the frontend and you want to add package-three and package-four to the compute nodes. Also, say the name of your roll is *diamond*.

First, in your development source tree, you'll create the file `roll/diamond/nodes/diamond-frontend.xml` which looks like:

```
<?xml version="1.0" standalone="no"?>

<kickstart>

<description>
The Diamond Roll.
</description>

<changelog>
</changelog>

<package>package-one</package>
<package>package-two</package>

<post>
<!-- post configuration scripts go here -->
</post>

</kickstart>
```

Then you'll create the file `roll/diamond/nodes/diamond-compute.xml` which looks like:

```
<?xml version="1.0" standalone="no"?>

<kickstart>

<description>
The Diamond Roll.
</description>

<changelog>
</changelog>

<package>package-three</package>
<package>package-four</package>

<post>
<!-- post configuration scripts go here -->
</post>

</kickstart>
```

Now you'll create the graph file `roll/diamond/graphs/default/diamond.xml`. This file is what links the nodes files `diamond-frontend.xml` and `diamond-compute.xml` into the full kickstart graph that is created on the frontend during the frontend's installation.

The graph file for the diamond roll:

```
<?xml version="1.0" standalone="no"?>

<graph>

<description>
The Diamond Roll.
</description>

<changelog>
</changelog>

<edge from="frontend">
 <to>diamond-frontend</to>
</edge>

<edge from="compute">
 <to>diamond-compute</to>
</edge>

</graph>
```

2.1.3.2. How it Works

During the installation of the frontend, the frontend's kickstart graph will be generated after all the rolls have been inserted by the user. That is, when the user answers 'No' to the the install screen that asks 'Do you have a Roll CD/DVD?', the installer will traverse the graph files and node files from all the rolls that were added to the frontend and build a RedHat-compliant kickstart file based on which nodes it traverses. The installation of the frontend is dictated by this generated kickstart file.

In the above example, for a frontend installation, the node file *diamond-frontend* will be traversed (and therefore included in the frontend's kickstart file), but the node file *diamond-compute* will not.

But, after the frontend is installed and when compute nodes are being installed, the node file *diamond-compute* will be traversed (and therefore included in the computes' kickstart file), but the node file *diamond-frontend* will not.

2.1.4. RPMS Directory

If you have any pre-built RPMS (for example, an RPM that you downloaded), you should place them in the `RPMS` directory.

All RPMS that you build from scratch (which will be built out of the files found under the `src` directory) will be automatically put in this directory.

2.1.5. SRPMS Directory

If you have any pre-built SRPMS (for example, an SRPM that you downloaded), you should place them in the `SRPMS` directory.

All SRPMS that you build from scratch (which will be built out of the files found under the `src` directory) will be automatically put in this directory.

2.1.6. src Directory

If your roll requires RPMs to be built from source, you'll place the tarballs under the `src` directory. If your roll doesn't need to build any source code, that is, it only includes downloaded pre-built RPMS, then the `src` directory is not needed and you should skip this step.

Looking at the Intel Roll, you'll see the file: `roll/intel/src/Makefile`. Copy this file into your `src` directory. You will not need to modify this file -- it simply drives the source code building process.

Now, in the Intel Roll, you'll see the directory: `roll/intel/src/mpich-eth`. Inside that directory, you'll see: `Makefile, mpich-1.2.5.2.tar.gz, mpich.spec.in and patch-files`.

The file `mpich-1.2.5.2.tar.gz` is the tarball as downloaded from the MPICH web site.

The directory `patch-files` is used to patch some of the files inside the mpich tarball. If you don't need to patch any files, ignore this directory.

The file `mpich.spec.in` is a RedHat spec file that directs `rpmbuild` on how to assemble the RPM. For an in-depth description of spec files, see Maximum RPM[3].

The file `Makefile` drives the building of the RPM.

Copy the files `Makefile and mpich.spec.in` to your `src` directory and experiment with them in order to build an RPM for your source code.

2.2. Building Your Roll

2.2.1. Making a Roll ISO

To build an ISO image of your roll, go the the directory: `roll/<roll name>` and execute:

```
# make roll
```

This will start a process that goes into the `src` directory and creates RPMS for each software package (as specified by their respective spec files), copies the newly built RPMs into the directory `RPMS`.

Then a directory named `iso` is created. It will be the structure of the ISO, that is, `mkisofs` looks into this directory and makes an ISO image based on it.

When you look at your ISO image, you'll see a file of the form: `roll-<roll name>-kickstart-3.1.0-0.noarch.rpm`. This RPM was automatically built for you and it contains the files found in the `nodes` directory and the `graphs` directory. This file is what the installer looks at in order to recognize your roll and properly install it.

After you successfully build your roll, you should see a file of the form: `roll-<roll name>-3.1.0-0.<arch>.iso`. This is the ISO image for your roll.

For example, the Intel Roll's ISO image for the i386 architecture is named: `roll-intel-3.1.0-0.i386.iso`.

Notes

1. http://www.rocksclusters.org/rocks-documentation/4.1/customization-mirroring.html
2. http://www.rocksclusters.org/rocks-documentation/4.1/anonymous-cvs.html
3. http://www.rpm.org/max-rpm/

Chapter 3. Kickstart Graph

In this chapter we give the details of the Rocks XML kickstart graph language, used to generate kickstart files. Here you will find a reference for Kickstart Node XML tags, edge tags, and order tags.

We also will discuss the roll of Linux distributions in Rocks.

3.1. Kickstart XML

Rocks generates Kickstart files for compute nodes dynamically using a structure called the Kickstart Graph. This graph is made from edges and nodes, both described in an XML language. This section serves as a reference for the XML tags and attributes. It is not an XML DTD. The XML language is specified by the `KPP`, `KGEN`, and `KCGI` applications.

The kickstart graph is specified by G=(V,E) where G is the graph, V is the set of nodes, and E is the set of edges that connect nodes. A traversal of the kickstart nodes and edges makes a full kickstart file used to specify the software and configuration for a node. Edges are described in "graph" files, while each node is specified in a "node" file.

When the order of nodes' contribution to the kickstart file matters, we use special edges made with <order> tags. We describe these tags in this section as well.

The <tagname> is given first, with a description. A list of attributes supported by the tag follows. All tags accept the **arch** attribute, which may be a list: "x86_64", or "i386, ia64". The tag is ignored if the client's architecture does not match this list.

3.1.1. Node XML

- **<kickstart>**

  Wraps an XML node file.

  roll

  Optional. Specifies the roll this node belongs too. Default "base".

- **<description>**

  Text description of what this node does.

- **<changelog>**

  Text description of the changes made to this node file. Generally a "Log" CVS directive.

- **<package>**

  Specifies a single RPM package. Includes only the RPM name, not its version or release number.

roll

Optional. Specifies the roll this package belongs too. Default "rocks".

type

Optional. Value is "meta". Used to describe RedHat meta packages. Examples are "Base", "Emacs", "GNOME".

disable

Optional. If this attribute exists, the RPM will not be installed, even if it is specified by a RedHat meta package.

- **<include>**

KPP replaces this tag with the contents of the file specified. Done primarily to enable syntax-friendly editing of script and language files.

file

Required. The file to include.

mode

Optional. Value is (quote|xml). If "quote", XML characters in the file are escaped appropriately. Default is "quote".

- **<var>**

In "read" form, KPP substitutes this tag with its value. In "write" form, KPP creates a variable for later reading. The value is generally found in the `app_globals` table of the Cluster database for compute nodes, and in the `site.xml` XML file (not ordinarily visible) for frontend installs. KCGI adds several additional variables, specified by the "Node" service name.

name

Required. The name of this variable. Format is generally `Service_Component`, where Service and Component names are used to match values in the app_global table of the database. Examples are `Kickstart_PublicDNSDomain`, and `Info_ClusterName`.

val

Optional. Sets the value of this variable. A "write" form.

ref

Optional. Allows you to set the value of this variable to the value of the variable specified here. A "write" form.

- **<eval>**

Replaced with the output of the script specified between these tags. The script is run on the host generating the kickstart file (generally the frontend).

shell

Optional. Specifies the script interpreter to use. If shell is "python", the `Applets` directory is added to the Python include path. This allows easy inclusion of python modules. Default "sh".

All `var` variables are made available to the script in the form of environment variables.

mode

Optional. Value is (quote|xml). If value is "quote", KPP escapes any XML characters in the shell output. Default is "quote".

- **<post>**

Wraps the post section of this node. Configuration is generally carried out in the post section, making this a popular tag. The commands specfied here correspond to an RPM post section, and they are executed on the client machine (not on the frontend, in contrast to the <eval> tag).

arg

Optional. Passed straight to anaconda. Common values are `--interpreter /usr/bin/python` and `--nochroot`.

- **<pre>**

Wraps the pre section commands. Run before package installation, in contrast to commands from the post section.

arg

Optional. Same semantics as the post section attribute.

- **<file>**

Wraps the contents of a file.

name

Required. Specifies the name of this file, a full path.

mode

Optional. Value is (create|append). If "create", file is created. If "append", the contents are appended to the end of an existing file. If target file does not exist, it will be created. A record of the change is kept in a RCS repository in the same directory as the file. Default is "create".

owner

Optional. The user.group that owns this file. Can be specified either as names "root.root" or numbers (guids) "0.0".

perms

Optional. The permissions of this file. The value of this argument is passed to the "chmod" command, and accepts the same format. Examples are "0755" or "a+rx".

vars

Optional. Value is "literal" or "expanded". If "literal" no variable or backtick expansion is done on the contents of the file. If value is "expanded", standard shell variable expansion is performed, as well as running commands quoted with backticks. Default is "literal".

expr

Optional. Specifies a command whose output is placed in the file. Should only be used as a singleton tag:

```
<file name="/etc/rocks.zone" expr="/opt/rocks/dbreport dns config"/>
```

3.1.1.1. Kickstart Main Section

These tags specify commands in the "main" section of a kickstart file. Each of these tags are wrapped in <main> tags. They appear in node XML. Only the tags normally used in a cluster appliance kickstart file are presented here; for a full reference see Red Hat Linux 7.3: The Official Red Hat Linux Customization Guide[1].

<main>

- **<auth>**

  Commands for the authentication of the system. Default "--useshadow --enablemd5 --enablenis --nisdomain rocks".

- **<zerombr>**

  If "yes" any invalid partition tables found on disks are initialized. Default "yes".

- **<bootloader>**

  Specifies the bootloader arguments. Default "--location=mbr"

- **<interactive>**

  Optional. Allows for inspection and modification of the kickstart values given, via the snack screen interface. Default: present.

- **<url>**

  Specifies the installation method with the `--url` argument. Default for compute nodes "--url http://$KickstartHost/$KickstartBasedir/$Node_Distribution" where variables are specified with the <var> construct.

  Default for frontend nodes: "--url http://127.0.0.1/mnt/cdrom"

- **<network>**

  Arguments for network configuration.

- **<lang>**

  The installation language to use. Default "en_US".

- **<langsupport>**

  The languages to install on the system. Example `<langsupport> --default en_US fr_FR</langsupport>`.

  Default "--default en_US".

- **<part>**

  Define a partition on the drive. See usage and warnings in Rocks Compute Node Partitioning[2].

- **<clearpart>**

  Clear all disk partitions on all drives.

- **<keyboard>**

  Sets the system keyboard type. Default "us".

- **<mouse>**

  Specifies the system mouse type. Default "none".

- **<timezone>**

  Required. Sets the system timezone. Default "--utc GMT".

- **<text>**

  If present, perform the kickstart installation in text mode. Default: present.

- **<install>**

  If present, perform a fresh install (not an upgrade). Default: present.

- **<reboot>**

  If present, automatically reboot after installation. Default: present.

3.1.2. Graph XML

Edges in the kickstart graph are specified with the XML tags below. Order tags give control of the graph traversal (the order nodes appear in the final kickstart file).

Both the edge and order tags appear in kickstart graph files.

- **<graph>**

  Wraps an XML graph file.

  roll

  Optional. Specifies the roll this node belongs too. Default "base".

- **<description>**

  Text description of what this part of the graph does.

- **<changelog>**

  Text description of the changes made to this node file. Generally a "Log" CVS directive.

- **<edge>**

  Specifies an edge in the kickstart graph, links graph nodes together. Can be specified as a singleton tag:

  `<edge from="client" to="java"/>`, or a standard tag that wraps <to> or <from> tags.

  arch

  Optional. Specifies which architectures should follow this edge. Same format as arch attribute in node files. The edge is ignored if the client's architecture does not match this list.

  gen

  Optional. Value is "kgen". Specifies which kickstart generator should follow this edge. Default "kgen".

from

Optional. Specifies the end of this edge, a node name.

to

Optional. Specifies the beginning of this edge, a node name.

- **<to>**

  Wraps a node name. Specifies the end of a directed edge in the graph. Used inside an edge tag with the "from" attribute:

```
<edge from="slave-node">
 <to>xinetd</to>
 <to>rsh</to>
</edge>
```

  arch

  Optional. Specifies which architectures should follow this edge. The entire edge is ignored if the client's architecture does not match this list.

- **<from>**

  Wraps a node name. Specifies the beginning of a directed edge. Used like "to" tag.

  arch

  Optional. Specifies which architectures should follow this edge. The entire edge is ignored if the client's architecture does not match this list.

- **<order>**

  Specifies a ordering between nodes in the graph. While the <edge> tags specify a "membership" in the kickstart file, the <order> tags give an "ordering" between nodes.

  The ordering is affected by a topological sort of nodes using order edges. While the kickstart graph allows cycles, the set of order tags must specify a directed-acyclic graph (DAG). Any nodes not touched by an order edge have a random order in the resultant kickstart file.

  Can be used to wrap <head> and <tail> tags in the same fashion as the <to> and <from> tags for "edge".

  arch

  Optional. Specifies which architectures should follow this edge. Same format as arch attribute in node files. The edge is ignored if the client's architecture does not match this list.

  gen

  Optional. Value is "kgen". Specifies which kickstart generator should follow this edge. Default "kgen".

head

Optional. Specifies the beginning of this edge, a node name. Special node name "HEAD" is allowed, which specifies the node is placed first. Ordering among nodes with HEAD ordering is undefined.

tail

Optional. Specifies the end of this edge, a node name. Special name "TAIL" is allowed, which specifies the node be placed last. Ordering among nodes with TAIL ordering is undefined.

3.2. Rocks Linux Distributions

A distribution is a collection of RPM software packages and an installation environment used to initialize a Linux machine. The distributions in Rocks are supersets of the RedHat Linux distribution, and have the same directory structure.

While RedHat has one type of distribution, Rocks has three:

- **Mirror.** A collection of RPMS. The authoritative repository of software for the cluster. RPMS from each installed roll are stored here. In a mirror, rolls are kept separate from each other based on version, architecture, and name.
- **Distribution.** An extension of a Mirror. Contains RPMS packages, the Rocks kickstart generation XML files, and a patched (extended) version of the installation environment. A distribution is always generated by the `rocks-dist dist` command run on the frontend.

  All cluster appliances use a distribution for installation. All roll RPMS regardless of origin or architecture are stored together in a distribution.

- **Pristine Distribution.** A distribution without any Roll RPMS. Contains only RPMS and kickstart nodes from Rocks base. Used to make CDs, and the Rocks Base distribution for WAN kickstart. This type of distribution is made with `rocks-dist --pristine dist`.

When building a distribution, the rocks-dist command retrieves RPM packages from several locations.

- The mirror. This is specified in the rocks-dist config file (XML format): `/opt/rocks/etc/rocks-distrc`.
- The `/usr/src/redhat` directory and its children. This is where the rpm system puts freshly built packages.

If rocks-dist finds two or more packages with the same name, it always chooses the RPM with the latest timestamp according to the filesystem.

Notes

1. http://www.redhat.com/docs/manuals/linux/RHL-7.3-Manual/custom-guide/s1-kickstart2-options.html
2. http://www.rocksclusters.org/rocks-documentation/4.1/customization-partitioning.html

Chapter 4. Internals

This chapter presents techniques and structures we use to configure nodes in the Rocks system. We provide a detailed reference of the Cluster SQL database that tracks nodes and coordinates installation activities in the cluster. We also discuss the dbreport interface and its roll in generating UNIX configuration files.

4.1. DB Report

The Rocks SQL database configures UNIX services in the Rocks system using the *dbreport* facility. Configuration files are generated from the database with a "report", a process often initiated by `insert-ethers`. This section presents the files under dbreport control.

These files are under dbreport control. Since they are dynamically generated, they should not be edited by hand. To alter the form of these files, you must edit the dbreports themselves, contained in the `rocks-dbreport` RPM package. In certain cases, a dbreport will respect a local configuration file and include its contents in the final output.

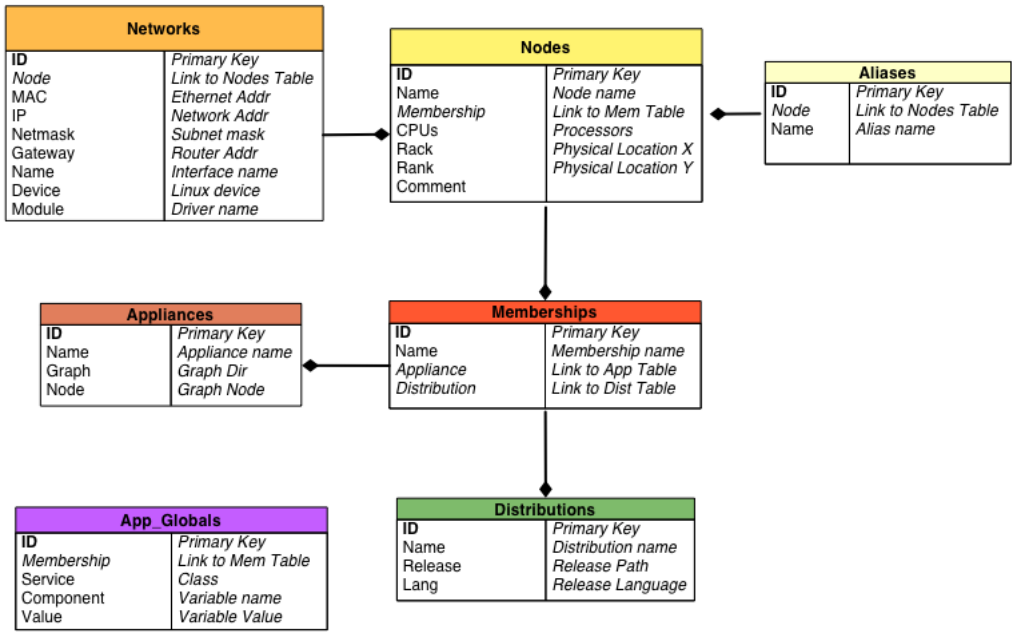
- **dhcpd** Generates the DHCP daemon's configuration file /etc/dhcpd.conf. Used by insert-ethers. No arguments.
- **dns** Generates the Bind named DNS server configuration files for the Private DNS domain in /var/named. Used by insert-ethers. Arguments are (config|reverse|zone). Respects the /var/named/rocks.domain.local and /var/named/reverse.rocks.domain.local files if present.
- **ethers** Generates a list of the ethernet MAC addresses for cluster nodes, along with their hostnames. No arguments.
- **gmond** Generates the config files for the Ganglia gmond daemon. Used by kickstart generators. Requires a node name argument.
- **hosts** Generates the /etc/hosts file. Used by insert-ethers. No arguments. Respects the /etc/hosts.local file if present.
- **list-pci** Queries what PCI cards are in which nodes of the cluster. Reserved for future use.
- **machines** Generates a simple list of compute node names, one per line. No arguments.
- **pvfs** Used to configure the parallel filesystem PVFS. No arguments.
- **resolv** Generates the /etc/resolv.conf file. Used in kickstart generation. Arguments are (public (default)|private). Public mode used for frontends, private for compute nodes.

4.2. The Cluster Database

This section describes the SQL Database Schema used by the Rocks system. The free *MySQL* DBMS server manages the schema in a single database named `cluster`. This database forms the backbone of the Rocks system, coordinating tasks as diverse as kickstart, node typing, configuration file building, and versioning.

4.2.1. Relational Schema

This diagram describes the database relations in a simplified standard notation.



4.2.2. Tables

A subset of the Information Engineering (IE) notation method will be used to describe the database tables. Primary keys are marked with an asterisk (*), and foreign keys are designated by (@). Attempts have been made to ensure this schema is at least in the Second Normal Form (2NF).

For each table, we present its structure followed by an explanation of its columns.

4.2.2.1. Nodes

Describes nodes in the cluster. A central table in the schema. The nodes table holds one row for each node in the cluster, including frontend, compute, and other appliance types. The node's location in the physical cluster (which rack it lies in on the floor) is specified in this table as well.

Table 4-1. Nodes

Field	Type
ID*	int(11)
Name	varchar(128)
Membership@	int(11)
CPUs	int(11)
Rack	int(11)
Rank	int(11)
Comment	varchar(128)

ID

A primary key integer identifier. Auto incremented.

Name

The name of the private network interface for this node. This violates the second normal form (2NF) of the schema (this name should only exist in the networks table), but serves as a hint for readability.

Membership

A link to the Memberships table; cannot be null. Specifies what type of node this is.

CPUs

The number of Processors in this node. Defaults to 1. Although this column violates the second normal form, it is more useful here than in a separate table.

Rack

The Y-axis coordinate of this node in euclidean space. Zero is the leftmost rack on the floor by convention. Note we only use a 2D matrix to locate nodes, the plane (Z-axis) is currently always zero.

Rank

The X-axis of this node in euclidean space. Zero is closest to the floor (the bottom-most node) by convention.

IP

The IPv4 Internet Protocol address of this node in decimal-dot notation.

Comment

A textual comment about this node.

4.2.2.2. Networks

The network interfaces for a node.

Table 4-2. Networks

Field	Type
ID*	int(11)
Node@	int(11)
MAC	varchar(32)
IP	varchar(32)
Netmask	varchar(32)
Gateway	varchar(32)
Name	varchar(128)
Device	varchar(32)
Module	varchar(128)

ID

A primary key integer identifier. Auto incremented.

Node

A link to the nodes table. A foreign key, cannot be null.

MAC

The 6-byte Media Access Layer address (Layer 2) of this interface in hex-colon notation like "a6:45:34:0f:44:99".

IP

The IPv4 Internet Protocol address of this interface in decimal-dot notation like "10.255.255.254".

Netmask

The subnet mask of the IP address. Specified like "255.0.0.0".

Gateway

The IP gateway or router address for this interface.

Name

The hostname for this network interface. The Rocks convention is `"compute-[Rack]-[Rank]"` for compute nodes.

If the device name is "eth0" this is the "private" network interface for the node. The interface name is placed in the .local domain and served via DNS. All other hostnames must be fully-qualified.

Device

The Linux device name for this NIC. The private (primary) interface always has the name "eth0".

Module

The Linux device driver name for this interface. Hardware specific.

4.2.2.3. App_Globals

This table contains Key=Value pairs used for essential services such as Kickstart. Examples include the Keyboard layout, Public Gateway, Public Hostname, DNS servers, IP mask, Cluster Owner, Admin email, etc.

Table 4-3. App_Globals

Field	Type
ID*	int(11)
Membership@	int(11)
Service	varchar(64)
Component	varchar(64)

Field	Type
Value	text

ID

A primary key integer identifier. Auto incremented.

Membership

A foreign key that references the `ID` column in the Membership table.

Service

The service name that will use this KEY=VALUE pair. Examples are “Kickstart” and “Info”. This is essentially the first part of a two-level naming scheme.

Component

The second level name. Together the Service and Component names correspond to the *name* attribute of the `<var name="Service_Component"/>` XML tag used in the kickstart generation process.

Value

The value of this row. Can be any textual data.

4.2.2.4. Memberships

This table specifies the distribution version and appliance type for a set of nodes. An alternative name for this table would be *groups*, however that is a reserved word in SQL. The memberships table names a group of nodes in the cluster and allows multiple memberships to tie into one appliance type.

Table 4-4. Memberships

Field	Type
ID*	int(11)
Name	varchar(64)
Appliance@	int(11)
Distribution@	int(11)
Compute	enum(’yes’,’no’)

ID

A primary key integer identifier. Auto incremented.

Node

The name of this membership. A type of node in the cluster, like "Frontend", "Compute", "Power Unit" or similar. The software installed on nodes in a given membership is defined by an appliance ID.

Appliance

A foreign key that references the `ID` column in the Appliances table. Helps define the software installed on nodes in this membership, and therefore their behavior.

Distribution

A foreign key that references the `ID` column in the Distributions table. The second key used to define the software for nodes in this membership.

Compute

Either "yes" or "no". Specifies whether this type of node will be used to run parallel jobs.

4.2.2.5. Appliances

This table defines the available appliance types. Each node in the cluster may classify itself as a single appliance type. The `Graph` and `Node` attributes define the starting point in the Rocks software configuration graph (See Graph XML), which wholly specifies the software installed on a node.

Table 4-5. Appliances

Field	Type
ID*	int(11)
Name	varchar(32)
ShortName	varchar(32)
Graph	varchar(64)
Node	varchar(64)

ID

A primary key integer identifier. Auto incremented.

Name

The name of this appliance. Examples are “frontend” and “compute”.

ShortName

A nickname for this appliance.

Graph

Specifies which software configuration graph to use when creating the kickstart file for this appliance. The default of `default` is generally used.

Node

Specifies the name of the root node in the configuration graph to use when creating the kickstart file for this appliance. The software packages for this appliance type is wholly defined by a traversal of the configuration graph beginning at this root node.

4.2.2.6. Distributions

This table connects a membership group to a versioned Rocks distribution, and plays an important role in the Rocks kickstart generation process. The `Release` relates to the RedHat distribution version, e.g. “7.2”, while the `Name` specifies where to find both the Rocks configuration tree and RPM packages. The location of these resources will be under the `/home/install/[Name]/[Release]/` directory.

Table 4-6. Distributions

Field	Type
ID*	int(11)
Name	varchar(32)
Release	varchar(32)
Lang	varchar(32)

ID

A primary key integer identifier. Auto incremented.

Name

Specifies where to find the Rocks configuration tree graph files. The *Name* field of the configuration graph located in the `/home/install/[Name]/[Release]/` directory.

Release

Gives the the RedHat distribution version this configuration tree is based on , e.g. “7.2”. The *Release* field in the graph location “`/home/install/[Name]/[Release]/`” directory.

Lang

The language of this distribution. A two-letter language abbreviation like "en" or "fr".

4.2.2.7. Versions

This table is intended to provide database schema versioning. It is reserved for future use.

Table 4-7. Versions

Field	Type
TableName	varchar(64)
Major	int(11)
Minor	int(11)

TableName

The name of a table in this database schema.

Major

The major version number of this table. Usually the first integer in the version string.

Minor

The minor version number of this table. The second integer in the version string.

4.2.2.8. Aliases

This table contains any user-defined aliases for nodes.

Table 4-8. Aliases

Field	Type
ID*	int(11)
Node@	int(11)
Name	varchar(32)

ID

A primary key integer identifier. Auto incremented.

Node

A foreign key that references the `ID` column in the Nodes table.

Name

The alias name. Usually a shorter version of the hostname.

Chapter 5. Monitoring Systems

In this section we describe the extensible Monitoring structures in Rocks.

Accurately and adequately monitoring a complex system such as a cluster is a difficult task. Our monitoring systems are for the most part modular, and accept user-defined additions. This section provides information to write your own cluster metrics and alarms.

5.1. Ganglia Gmetric

Rocks uses the *Ganglia* system to monitor the cluster. We have defined several extension structures to Ganglia that allow you to easily add metrics and alarms to your cluster monitoring system.

A single value to be monitored is called a *metric* in the Ganglia system. Metrics are measured on nodes of the cluster, and Ganglia communicates them to the frontend. This section describes how to write your own metric monitors.

5.1.1. Greceptor Metric

To define a new metric to monitor, we write a Metric module for the Rocks *greceptor* daemon. Throughout the monitoring chapter, we use "fan speed" as an example. This metric could read and monitor the speed of a CPU or chassis fan in a node. When the fan rpm is low, it could signal a malfunction that needs your attention.

1. To monitor a value, you must be able to read it programmatically. Write a script that returns the fan speed as a number. It can be written in any language, but should have the following behavior:

   ```
   # read-fan1-speed
   3000
   #
   ```

   The output should contain no units or any ancillary information, as they will be added later.

2. The greceptor metric module that calls your script must be written in Python. The listing below gives an example:

   ```
   import os
   import gmon.events

   class FanSpeed(gmon.events.Metric):

        def getFrequency(self):
            """How often to read fan speed (in sec)."""
            return 30

        def name(self):
            return 'fan1-speed'

        def units(self):
   ```

```
            return 'rpm'

        def value(self):
            speed = os.popen('read-fan1-speed').readline()
            return int(speed)


def initEvents():
     return FanSpeed
```

The class is derived from the Rocks `gmon.events.Metric` class. This module calls your 'read-fan1-speed' script to get the actual value. You may obtain your metric value directly using Python code if you like, just return its value at the end of the `value()` method.

Greceptor will call the `name(), value()` functions every 30 seconds or so. The actual frequency is uniformly random with an expectation of 30 seconds.

3. Make an RPM that places this file in:

   `/opt/ganglia/lib/python/gmon/metrics/fanspeed.py`

   Any name ending in *.py will work. Add the RPM to your roll and reinstall your compute nodes. See the `hpc-ganglia` package in the HPC roll for an example.

4. The metric value should be visible in the ganglia gmond output on the frontend, which can be inspected with:

   ```
   $ telnet localhost 8649
   ```

   If the metric value is an integer or float, a graph of its value over time is visible on the Ganglia web-frontend pages, in the Cluster or Host views.

5.1.2. Greceptor Listener

You may also define a greceptor module that listens for a particular Gmetric name on the Ganglia multicast channel. This is called a *listener*. Rocks uses listeners to implement 411, and setup the MPD job launcher ring.

We will use our fan-speed example. This process installs a listener for the metric we just defined.

1. Derive the listener from the Rocks class `gmon.events.Listener`. The name() method defines what metric name to listen for, it must match exactly. In this case we specify the 'fan1-speed' name from the previous example.

   ```
   import gmon.events

   class FanSpeed(gmon.events.Listener):
    """Does something when we hear a fan-speed metric."""
   ```

```
 fanthresh = 500

 def name(self):
  """The metric name we're interested in."""

  return "fan1-speed"


 def run(self, metric):
  """Called every time a metric with our name passes on the
  Ganglia multicast network in the cluster."""

  fanspeed = float(metric["VAL"])

  if fanspeed < self.fanthresh:
   self.getWorried()


 def getWorried(self):
  """Does something appropriate when the fan speed is too low."""

  # Some action
  pass


def initEvents():
 return FanSpeed
```

Greceptor will call the run() method whenever a 'fan1-speed' metric passes on the Ganglia multicast channel. The `metric` argument to run() is a Python dictionary keyed with the attribute names from a <METRIC> Ganglia metric. The source of the metric is given by `metric["IP"]`.

2. Make an RPM that places this file in:

   `/opt/ganglia/lib/python/gmon/listeners/fanspeed.py`

   Any name ending in *.py will work. Add the RPM to your roll and reinstall your compute nodes. See the `hpc-ganglia` package in the HPC roll for an example.

In the next section, we show how to make an alarm for a metric value using the Ganglia News construct.

5.2. News

Rocks has introduced a simple alert system called *Ganglia News* that generates RSS items (blog-style) for significant events in the cluster.

Each type of event has a Python module called a *Journalist* that can detect it. This section describes how to write your own journalist.

Journalists are run by a nightly cron job. In this example, we setup a journalist that writes an item when the fan-speed for a node falls below a threshold level. It relies on the gmetric shown in the previous section.

1. Derive the news module from the Rocks class `gmon.journalist`. The run() method is called for you during the cron job. You should call the recordItem() function with a RSS-snippet for any relevant items, and a name for each of them. The item name is generally a host IP address.

   The item name specifies a filename for the event, which is placed in:

   `/var/ganglia/news/[year]/[month]/[journalist-name]/[item-name].rss`

   The RSS news report is generated by concatenating all *.rss files in the current year and month. Therefore, news items will remain visible for the current month, then reset.

```
import gmon.journalist

class FanSpeed(gmon.journalist.Journalist):
 """A news collector that detects low fan speeds."""

 # When fan speed falls below this RPM level, we call it news.
 fanthresh = 500.0

 def name(self):
  return "fan1-speed"


 def run(self):
  c = self.getGanglia().getCluster()
  for h in c.getHosts():
   try:
    fanspeed = float(h.getMetricValue('fan1-speed'))

    if fanspeed < self.fanthresh:
     self.item(c, h, fanspeed)
   except:
    continue


 def item(self, cluster, host, fanspeed):

  s = '<item>\n' \
    + ' <description>\n' \
    + '%s. Node %s fan is malfunctioning: its speed is %s rpm.\n' \
     % (self.getDate(), host.getName(), fanspeed) \
    + '(Threshold is %s))\n' \
   % (self.fanthresh) \
    + ' </description>\n' \
    + ' <link>%s</link>\n' % self.getHostPage(cluster, host) \
    + ' <pubDate>%s</pubDate>\n' % self.getTime() \
    + '</item>\n'
```

```
  self.recordItem(host.getIP(), s)


def initEvents():
 return FanSpeed
```

We iterate over all nodes in cluster, looking for bad fans. The `recordItem(), getTime(), getHostPage()` methods are provided by the Journalist class. The RSS conforms to the 2.0 specification.

2. Make an RPM that places this file in:

   `/opt/ganglia/lib/python/gmon/news/fanspeed.py`

   Any name ending in *.py will work. Add the RPM to your roll and reinstall your compute nodes. See the `ganglia-news` package in Rocks Base for an example.

Point your RSS browser to the "News" link of your cluster (available on the cluster homepage). You should see a News item for each node that has a slow fan.

Bibliography

Papers

Leveraging Standard Core Technologies to Programmatically Build Linux Cluster Appliances , Mason J. Katz, Philip M. Papadopoulos, and Greg Bruno, April 2002 , CLUSTER 2002:[1] IEEE International Conference on Cluster Computing , (PDF)[2] .

NPACI Rocks: Tools and Techniques for Easily Deploying Manageable Linux Clusters , Philip M. Papadopoulos, Mason J. Katz, and Greg Bruno, Submitted: June 2002 , Concurrency and Computation: Practice and Experience[3] Special Issue: Cluster 2001 , (PDF)[4] (PostScript)[5] .

Wide Area Cluster Monitoring with Ganglia , Federico D. Sacerdoti, Mason J. Katz, Matthew L. Massie, and David E. Culler, December 2003, IEEE International Conference on Cluster Computing, Hong Kong , (PDF)[6] .

Notes

1. http://www-unix.mcs.anl.gov/cluster2002/
2. http://www.rocksclusters.org/rocks-documentation/4.1/papers/clusters2002-rocks.pdf
3. http://www3.interscience.wiley.com/cgi-bin/jtoc?Type=DD&ID=88511594
4. http://www.rocksclusters.org/rocks-documentation/4.1/papers/concomp2001-rocks.pdf
5. http://www.rocksclusters.org/rocks-documentation/4.1/papers/concomp2001-rocks.ps
6. http://www.rocksclusters.org/rocks-documentation/4.1/papers/cluster2003-ganglia.pdf