Nuts and Bolts: Developing Rolls

How do you reliably add and configure (complex) software in a cluster environment?

Rocks Philosophy

- We've developed a "cluster compiler"
  - XML framework + XML parser + kickstart (Jumpstart for Solaris) file generator
  - Source code + preprocessor + linker
- Think about "programming your cluster"
  - Not "administering your cluster"

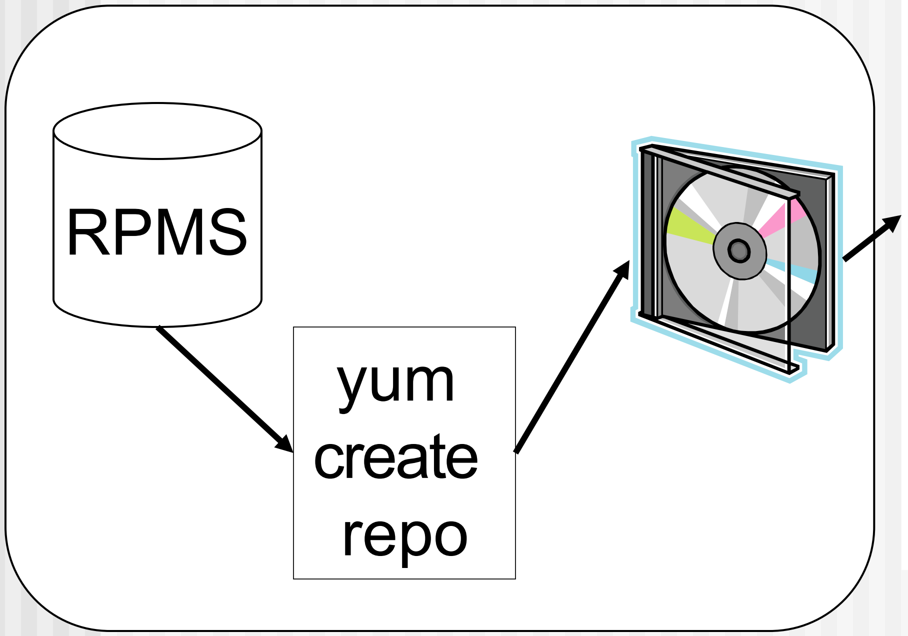
Purpose of Rolls

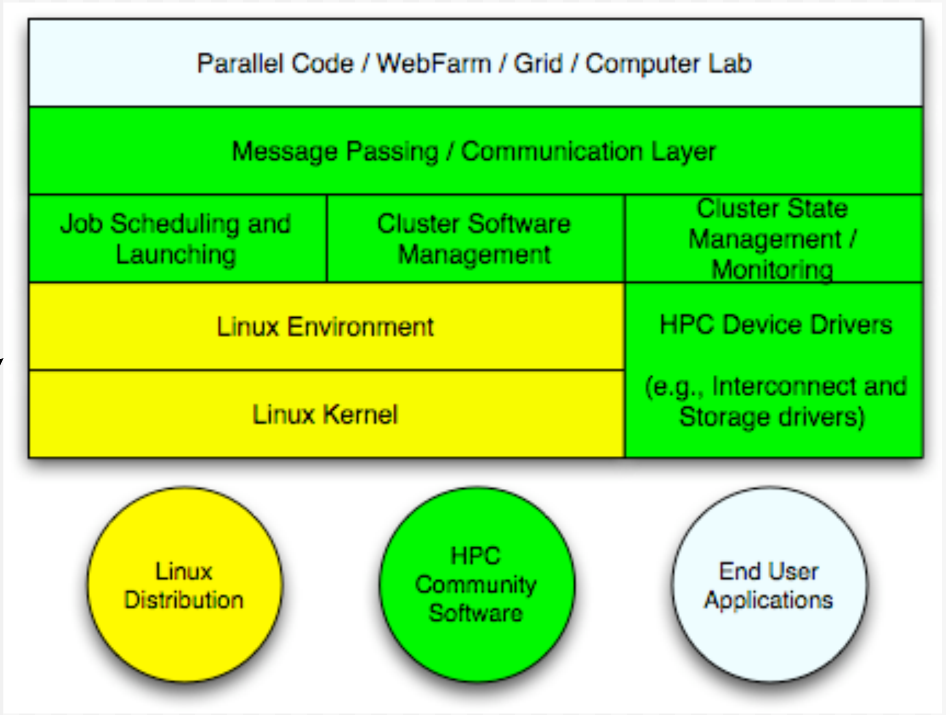
- Capture what the expert would do “by hand” for a particular subsystem and **automate it.**
- Enable others to **extend the system** to provide completely new functionality
- Make the clustered system **reliable and reproducible**

What’s in a Roll

1. Software Packages in native OS Distribution Format
   - RPMs for RHEL and Derivatives
   - PKG format for Solaris
2. Description of the set(s) packages to install on each node type (Appliance)
3. Configuration of installed software
   - What to do when a node is added/removed
   - Where is that Server?
   - What specific options should be included

As Delivered – OS Distributions are both Static and Monolithic

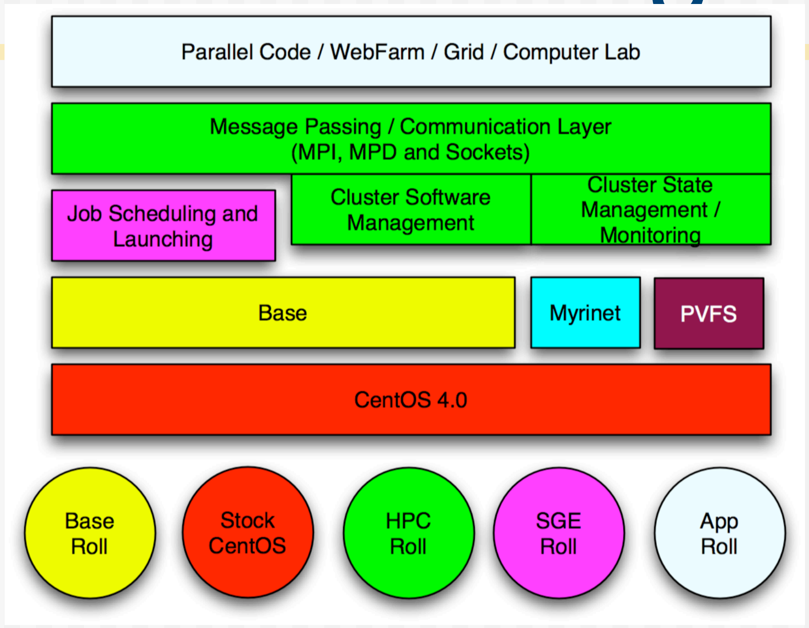



Parallel Code / WebFarm / Grid / Computer Lab
Message Passing / Communication Layer
Job Scheduling and Launching
Cluster Software Management
Cluster State Management / Monitoring
Linux Environment
Linux Kernel
HPC Device Drivers
(e.g., Interconnect and Storage drivers)
Linux Distribution
HPC Community Software
End User Applications

Bootstrapping on a Frontend (w/o a server in the sky)

- Problem: To make the frontend user-customizable at installation time, we needed a mechanism that could accept new packages
- And, we still wanted to leverage the RedHat installer (Anaconda)
  - We don’t want to be in the installer business
- Solution: Our implementation makes the RedHat installer “think” it is just installing a monolithic RedHat distribution

Just in time Package Repository


Parallel Code / WebFarm / Grid / Computer Lab
Message Passing / Communication Layer (MPI, MPD and Sockets)
Job Scheduling and Launching
Cluster Software Management
Cluster State Management / Monitoring
Base
Myrinet
PVFS
CentOS 4.0
Base Roll
Stock CentOS
HPC Roll
SGE Roll
App Roll

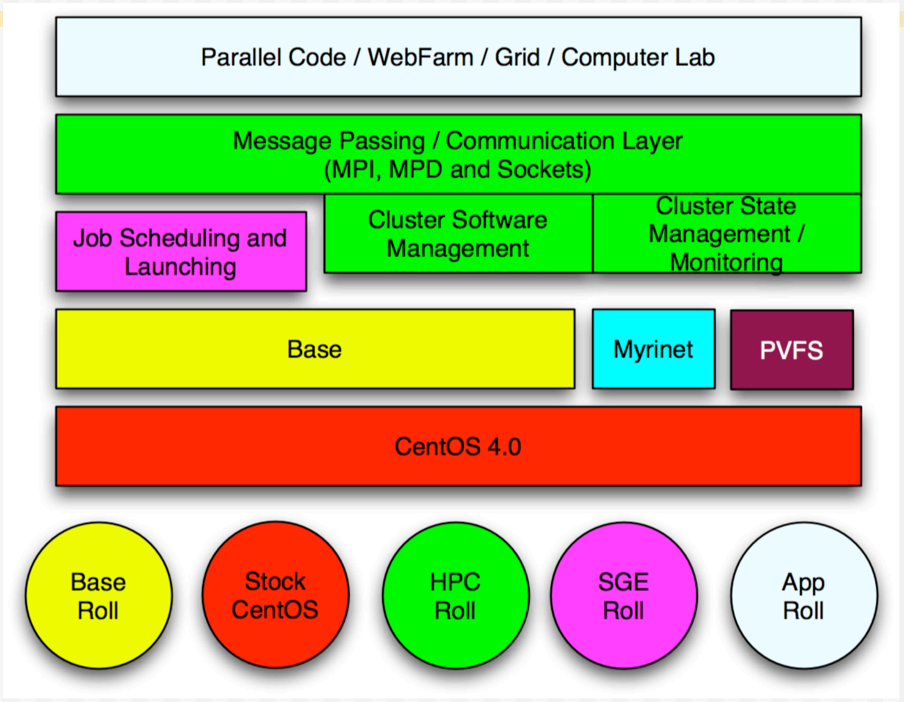
- How do you make all the packages above look like a monolithic distribution?
  - Easy! Just run “yum create repo” at release time! (And Burn a DVD)
- But, how do you do it when some of the above blocks are optional and/or unknown?
  - An “unknown” block is one produced after the release or by a third-party

Rocks Workhorse: Binding a New Distribution

- `rocks create distro`

- Called at install time after you have inserted all roll RPMS have been copied
- Called on the installed system, whenever an update to the distribution is required
- (Rolls can supply updated RPMs so that you can build an up-to-date system)

Rolls Function and Value


Parallel Code / WebFarm / Grid / Computer Lab
Message Passing / Communication Layer (MPI, MPD and Sockets)
Job Scheduling and Launching
Cluster Software Management
Cluster State Management / Monitoring
Base
Myrinet
PVFS
CentOS 4.0
Base Roll
Stock CentOS
HPC Roll
SGE Roll
App Roll

- Function: Rolls extend/modify stock RedHat
- Value: Third parties can extend/modify Rocks
  - Rolls can be optional.
  - Doesn’t solve does Roll X interoperate with Roll Y

Part I: Packages

Packages

- Rolls require packages to be in native OS format (RPM, Solaris pkg)
- The Good
  - Inspect software with native OS tools
  - Can install “by hand” using OS tools
- The Bad
  - You have to make your software into a package (only seems like a big hill)
  - Package Mechanisms can cause odd behavior

Our Philosophy on Packages

- We use packages as a transport
- Very little (none as a default) is done in the package %post section
  - This is what the Rocks node files are used for
- Stay away from explicitly creating “spec” files
- Make is your friend (ours too)

Make requirements

- We support building only a frontend node (that may change)
- Faith
  - There is large set of included make rules that allow us to quickly package software
  - You have to trust what is doing.

Different Ways For Packaging From Source

- Build software by hand, then point
  - Rocks create package at the directory
- Build an RPM Spec file
- Use the Rocks-supplied Make Infrastructure

Creating a Roll from a template

```
wget --reject "index.html*" -np -r -nH --cut-dirs=2 \
  http://fyp.rocksclusters.org/templates/5.1
```

Building an RPM

- Short story
  - Go to /export/site-roll/rocks/src/roll on a Rocks Frontend
  - Make a new roll from a ‘template’ roll
  - Download the source tarball
  - Update a description file (version.mk)
  - Execute: make rpm
    - Assumes tarball adheres to ‘configure, make, make install’

Using Rocks Make Environment

- Rocks frontend has the tooling to build rools
- cd /export/site-roll/rocks/src/roll/
- Let’s Make an RPM ---
- First, make a template for a new roll

```
#wget --reject "index.html*" -np -r -nH --cut-dirs=2 \
  http://fyp.rocksclusters.org/templates/5.1
# /opt/rocks/share/devel/src/roll/bin/make-roll-dir.py --
  name valgrind --version 3.3.0
# ls valgrind
graphs  Makefile  nodes  src  version.mk
```

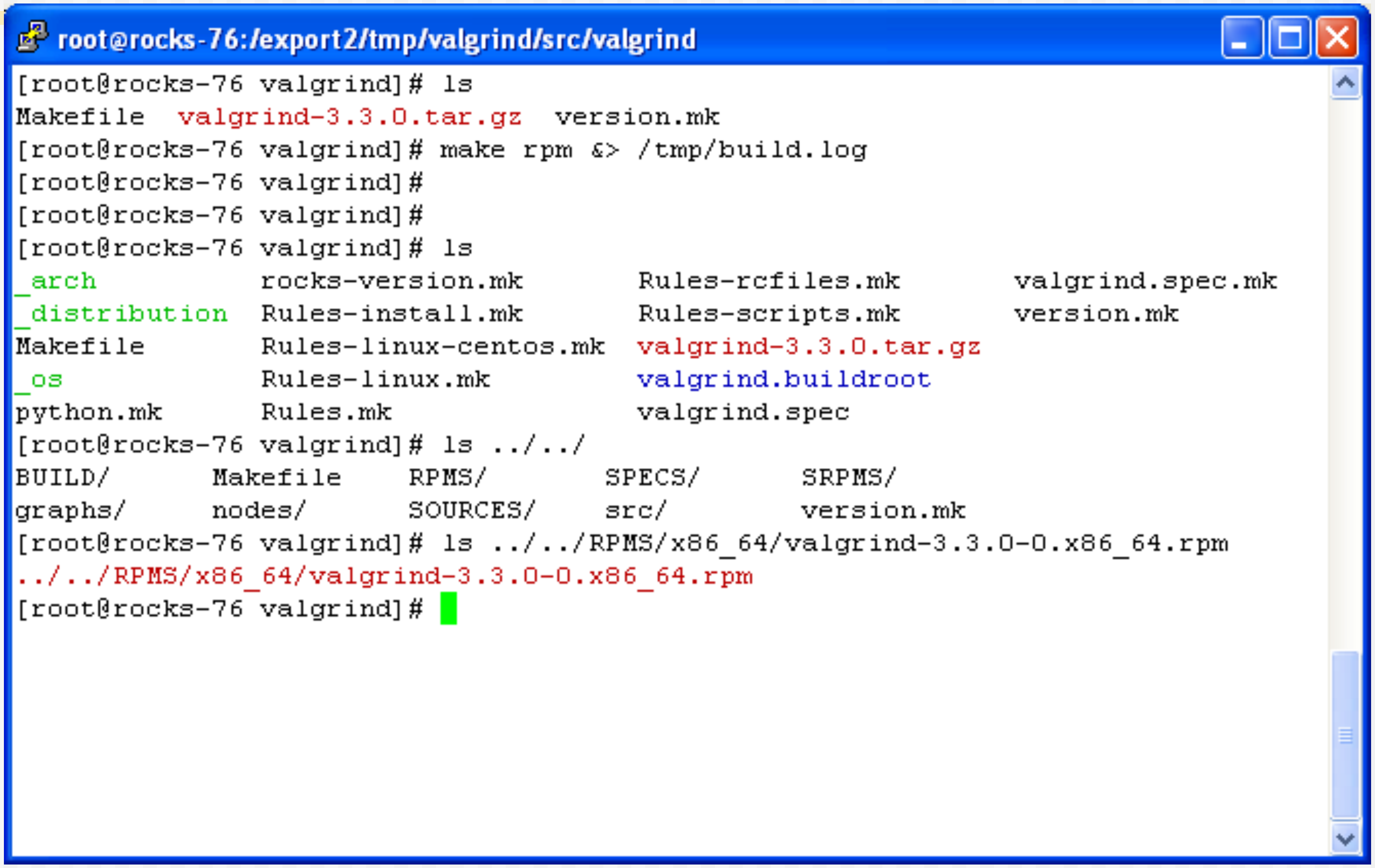
- **`valgrind/src/valgrind` has what you need to make an rpm**

src/valgrind – a working example

```
# cd valgrind/src/valgrind
# wget
   http://valgrind.org/downloads/valgrind-3.3.0.tar.bz2
# bunzip2 valgrind.*.bz2; gzip valgrind.tar
# rm *.spec.in
# edit version.mk so that
TARBALL_POSTFIX = tar.gz
# edit Makefile and undefine ROCKSROOT
# make package
# ls ../../RPMS/x86_64/valgrind-3.3.0-1.x86_64.rpm
../../RPMS/x86_64/valgrind-3.3.0-1.x86_64.rpm
```

That's it….. Works because valgrind is built using "./configure; make; make install"

Do it!



```
[root@rocks-76 valgrind]# ls
Makefile  valgrind-3.3.0.tar.gz  version.mk
[root@rocks-76 valgrind]# make rpm &> /tmp/build.log
[root@rocks-76 valgrind]#
[root@rocks-76 valgrind]#
[root@rocks-76 valgrind]# ls
_arch          rocks-version.mk       Rules-rcfiles.mk       valgrind.spec.mk
_distribution  Rules-install.mk       Rules-scripts.mk       version.mk
Makefile       Rules-linux-centos.mk  valgrind-3.3.0.tar.gz
_os            Rules-linux.mk         valgrind.buildroot
python.mk      Rules.mk               valgrind.spec
[root@rocks-76 valgrind]# ls ../../
BUILD/      Makefile    RPMS/       SPECS/      SRPMS/
graphs/     nodes/      SOURCES/    src/        version.mk
[root@rocks-76 valgrind]# ls ../../RPMS/x86_64/valgrind-3.3.0-0.x86_64.rpm
../../RPMS/x86_64/valgrind-3.3.0-0.x86_64.rpm
[root@rocks-76 valgrind]#
```

There is “magic” here

- We use the native OS Package as a transport
  - rpmbuild as the “package builder”
    - Needs an rpm spec file to drive it
    - We build a generic spec file automatically
- Standard RPM file tree needs the following directories to work properly

BUILD SOURCES SPECS

Step 0 of Magic Create a Standard SPEC File

- Creates a standard Redhat SPEC file, eg.

```
Source: valgrind-3.3.0.tar.gz
Buildroot: `pwd`/valgrind.buildroot
%prep
    (unpack the tarball created in step 1)
%build
    (call make build)  ← Makefile is the src/
  valgrind Makefile

%install
    (call make install)
```

Step 1 of Magic – Create a Source File to go in SOURCES

1. Automatically creates a tarball of the current directory. Calls this <name>-<version>.tar.gz

2. Copies this file into the SOURCES Directory

   * contains this complete directory including the “real” software tarball

Making the SOURCES file --

```
[root@rocks-76 valgrind]# tree
.
|-- Makefile
|-- valgrind-3.3.0.tar.gz
`-- version.mk
```

valgrind-3.3.0.tar.gz

<rollname>/SOURCES/valgrind-3.3.0.tar.gz

Step 3 of Magic: The BUILD Directory

untar

<rollname>/BUILD/valgrind-3.3.0

valgrind-3.3.0.tar.gz

```
[root@rocks-76 BUILD]# tree
.
`-- valgrind-3.3.0
    |-- Makefile
    |-- valgrind-3.3.0.tar.gz
    `-- version.mk
```

SPEC File Calls This Makefile for %build, %install

You can intercept stages in the process

- Before the tarball is made
- Add patches, if needed
- Many examples, check any of the Rocks core rolls

When RPM goes Wrong

- Symptom – I’ve added an RPM an now my node installation is completely broken, what happened?
  - Observe: watch order that packages are installed on node (via rocks-console)
  - IF: packages are installed in alphabetical order then this package is breaking Anaconda’s dependency ordering
- Fix Need to Turn RPM Auto Requires/Provides off
  - In version.mk add
    - RPM.EXTRAS=AutoReqProv:No
  - Rebuild rpm

When RPM goes Wrong

- Symptom: RPM is stripping a (prebuilt) binary making it useless
- Solution: RPM hacking.
  - Redefine an RPM macro
  - Edit version.mk add

```
RPM.EXTRAS=%define __os_install_post /usr/lib/rpm/brp-compress
```

  - Rebuild rpm

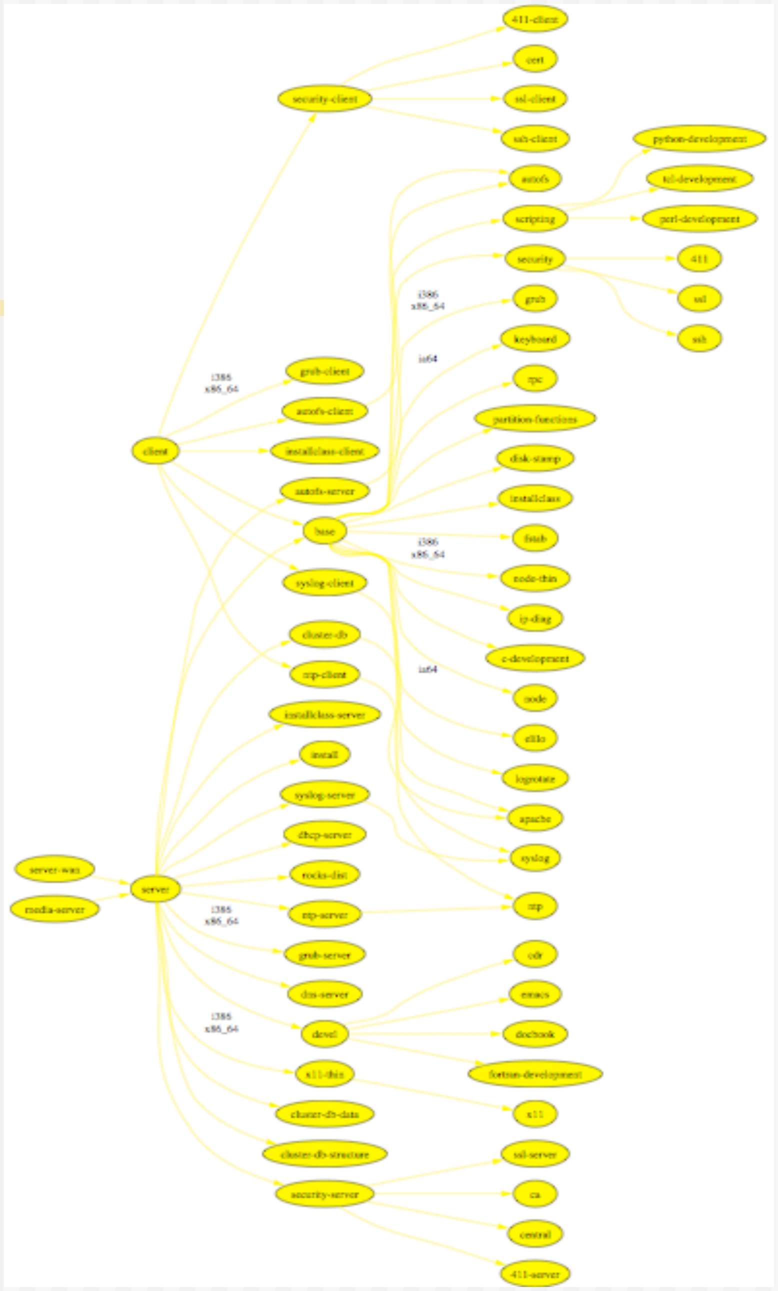
Part II: Defining Which Packages go Where

Graph Review

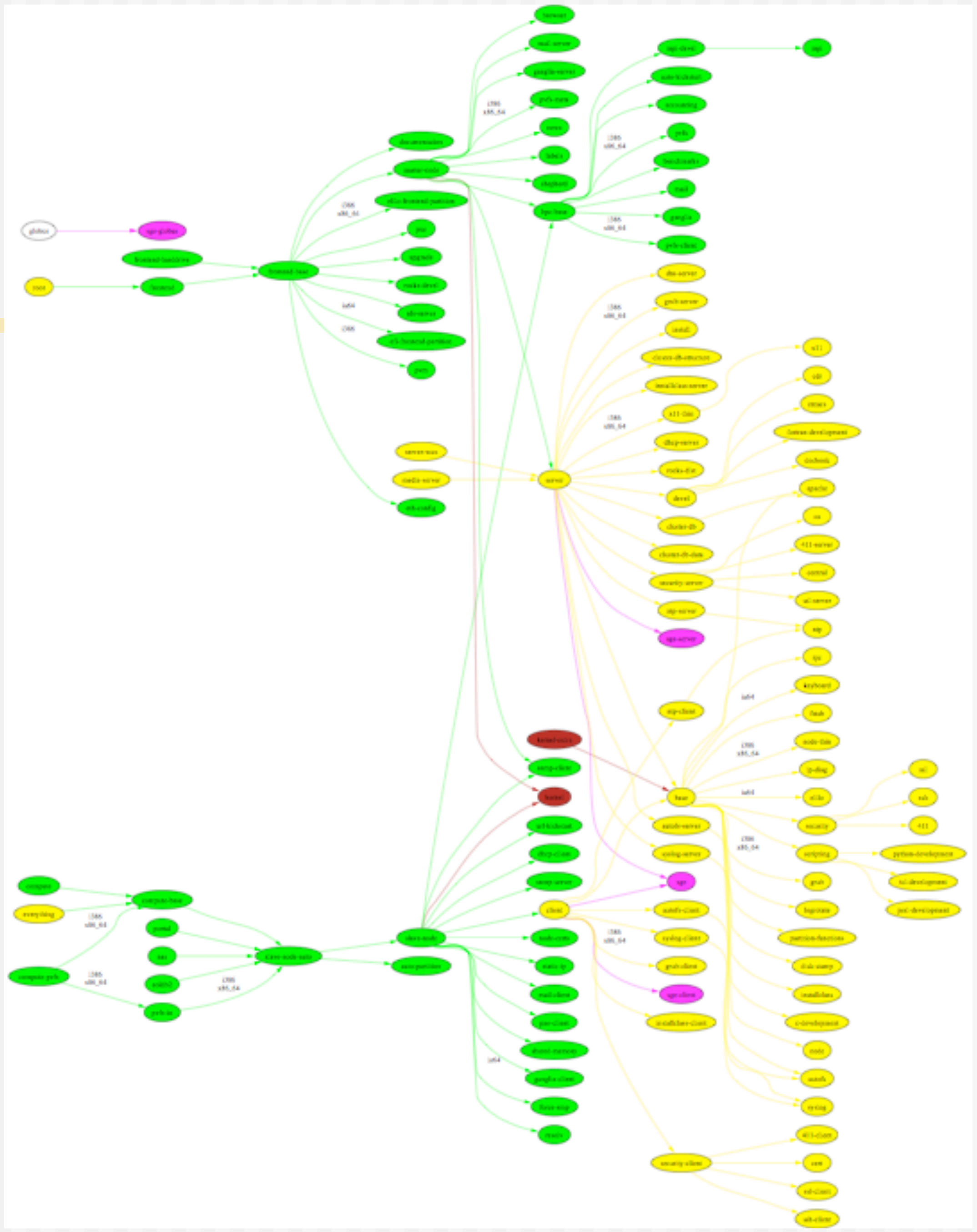
Install Rocks Base Graph

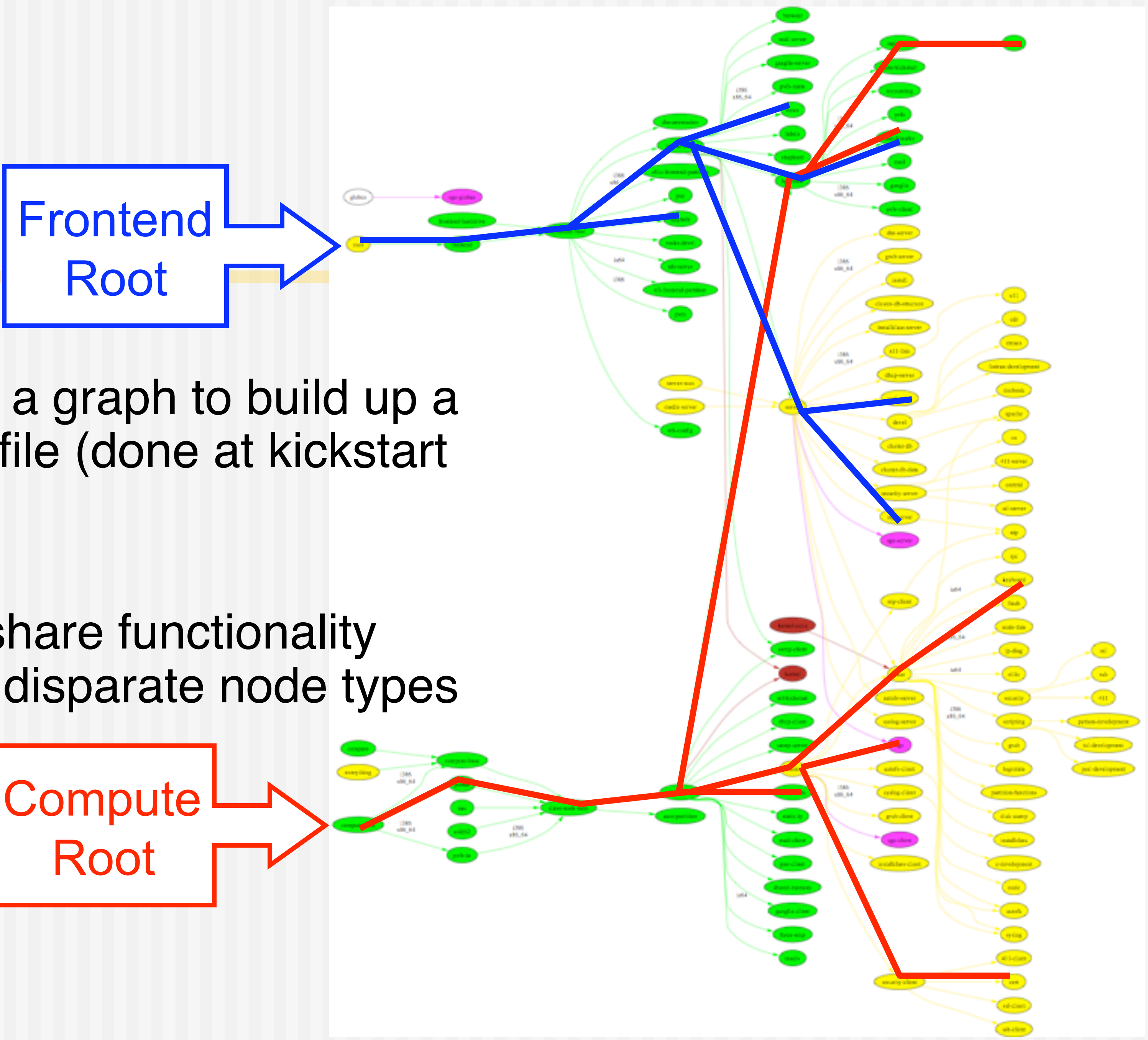
Basic Instructions that define all Rocks Appliances

Rolls have packages and graphs



Base + Rolls

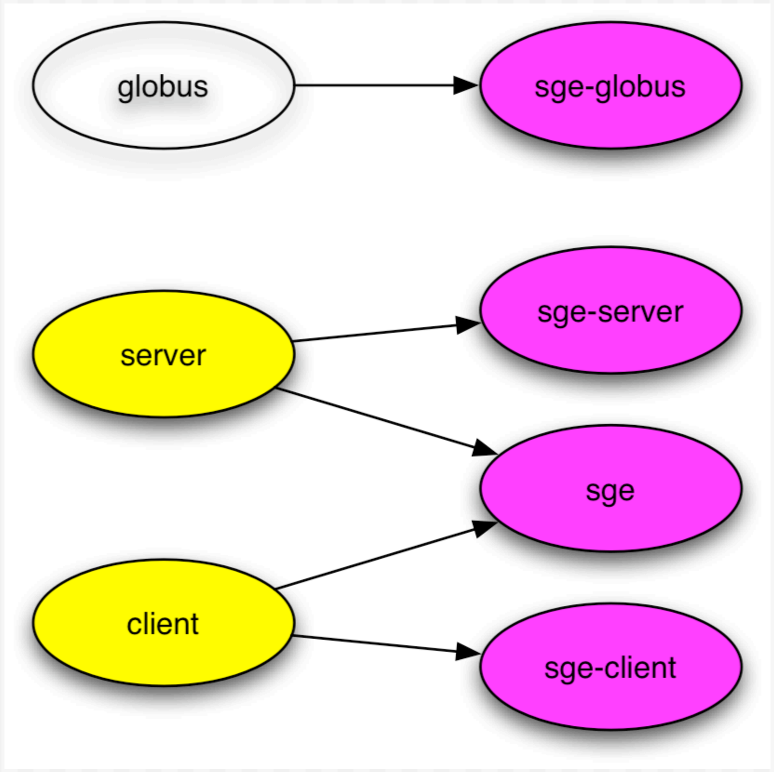




- Traverse a graph to build up a kickstart file (done at kickstart time)
- Flexible
- Easy to share functionality between disparate node types

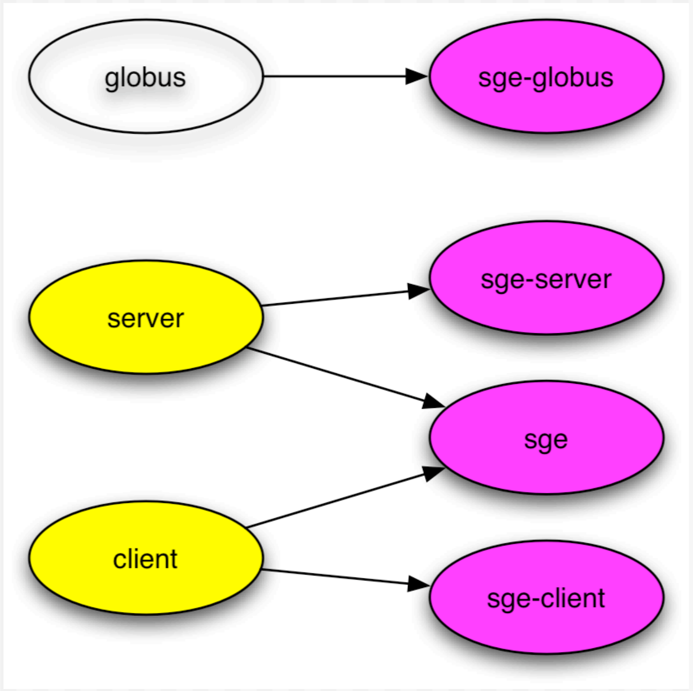
Use Graph Structure to Dissect Distribution

- Use ‘nodes’ and ‘edges’ to build a customized kickstart file
- Nodes contain portion of kickstart file
  - Can have a ‘main’, ‘package’ and ‘post’ section in node file
- Edges used to coalesce node files into one kickstart file



Why We Use A Graph

- A graph makes it easy to ‘splice’ in new nodes
- Each Roll contains its own nodes and splices them into the system graph file



XML Files

- We use XML files to define the **nodes** in the graph
  - What packages to install
  - What to do at <post> installation
- We also use XML files to define the **graph structure**

Node and Graph Dirs in Roll

```
[root@rocks-76 valgrind]#
tree
.
|-- Makefile
|-- graphs
|   `-- default
|       `-- valgrind.xml
|-- nodes
|   `-- valgrind.xml
|-- src
|   |-- Makefile
|   |-- usersguide
|   |   `-- valgrind
|       |-- Makefile
|       |--
valgrind-3.3.0.tar.gz
|       `-- version.mk
```

Unimaginative Names.

<package> Tag

- **`<package>java</package>`**
  - Specifies an RPM package. Version is automatically determined: take the *newest* rpm on the system with the name ‘java’.
- **`<package arch=“x86_64”>java</package>`**
  - Only install this package on x86_64 architectures
- **`<package arch=“i386,x86_64”>java</package>`**

```
<package>newcastle</package>
<package>stone-pale</package>
<package>valgrind</package>
```

%packages
newcastle
stone-pale
valgrind

Common Splitting of Node Files

- <roll>-server.xml
  - Things you install and configure only on Frontends
- <roll>-client.xml
  - Things you install and configure only on “client” nodes (eg. Compute, NAS, VM-containers, …)
- <roll>-common.xml
  - Things installed everywhere

Graph Edges: <edge>

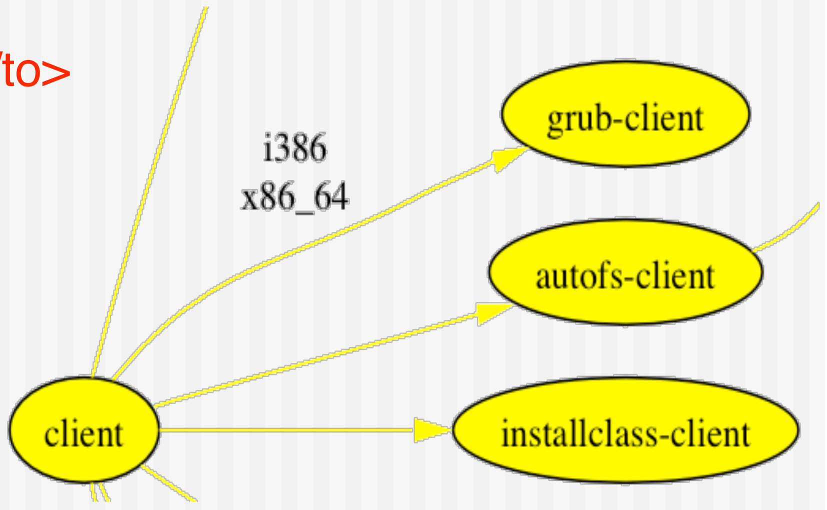
- <edge> attributes
  - from
    - Required. The name of a node at end of the edge
      - <edge from="base" to="autofs"/>
  - to
    - Required. The name of a node at the head of an edge
  - arch
    - Optional. Which architecture should follow this edge. Default is all.
  - gen
    - Optional. Which generator should follow this edge. Default is "kgen"

(IN 5.2 Edges can have conditionals based on attributes)

Graph Edges

```
<edge from="security-server" to="central"/>


<edge from="client">
    <to arch="i386,x86_64">grub-client</to>
    <to>autofs-client</to>
    <to>installclass-client</to>
</edge>
```



Graph Ordering

- Added recently to give us control over when node <post> sections are run
  - <order head="database">
    - <tail>database-schema</tail>
  - </order>
- *database* node appears before *database-schema* in all kickstart files.
- Special HEAD and TAIL nodes represent “first” and “last” (post sections that you want to run first/last)
  - <order head=“installclass” tail=“HEAD”/> BEFORE HEAD
  - <order head=“TAIL” tail=“postshell”/> AFTER TAIL

Graph Ordering: <order>

- <order> attributes
  - head
    - Required. The name of a node whose <post> section will appear BEFORE in the kickstart file.
  - tail
    - Required. The name of a node whose <post> section will appear AFTER in the kickstart file.
      - <order head="grub" tail="grub-server"/>
  - arch
    - Optional. Which architecture should follow this edge. Default is all.
  - gen
    - Optional. Which generator should follow this edge. Default is "kgen"

Valgrind Example: Connecting into the graph

```
# vi graphs/default/valgrind.xml ( and add:)
      <edge from="base">
            <to>valgrind</to>
       </edge>
```

This tells us that Valgrind should be on all appliances.

Roll is complete

- Can use it as a roll to build frontends
- A straightforward test if you have a compute node

```
# rocks add roll valgrind-*.iso
#rocks enable roll valgrind
# (cd /export/rocks/install; rocks create distro)
# rocks list host profile compute-0-0 | grep valgrind
# ./nodes/valgrind.xml (valgrind)
roll-valgrind-usersguide
valgrind
```

Where the art is: <post>

- Package Creation ranges from trivial to not-so-trivial
- Defining where packages go, some on this appliance, some on that. Straightforward
- But, the post section …

Nodes Post Section

- Scripts have minimal $PATH (/bin, /usr/bin)
- Error reporting is minimal
  - Write to personal log file if you need debugging
- Not all services are up. Network is however.
  - Order tag is useful to place yourself favorably relative to other services
- Can have multiple <post> sections in a single node

Nodes XML Tools: <post>

- <post> attributes
  - arch
    - Optional. Specifies which architectures to apply package.
  - arg
    - Optional. Anaconda arguments to *%post*
      - --nochroot (rare): operate script in install environment, not target disk.
      - --interpreter: specifies script language
      - <post arg="--nochroot --interpreter /usr/bin/python">

Post Example: PXE config

```
<post arch="x86_64,i386">
mkdir -p /tftpboot/pxelinux/pxelinux.cfg

<file name="/tftpboot/pxe../default">
default ks
prompt 0
label ks
		kernel vmlinuz
		append ks inird=initrd.img……
</file>
</post>
…

</post>
```

for an x86_64 machine:

```
cat >> /root/install.log << 'EOF'
./nodes/pxe.xml: begin post section
EOF
mkdir -p /tftpboot/pxelinux/pxelinux.cfg

…RCS…
cat > /tftpboot/pxe../default << EOF
default ks
prompt 0
…
EOF
..RCS…
```

A Real Node file: ssh

```
<kickstart>
         <description>
         Enable SSH
         </description>

         <package>openssh/package>
         <package>openssh-clients</package>
         <package>openssh-server</package>
         <package>openssh-askpass</package>
<post>

<file name="/etc/ssh/ssh_config">
Host *
        CheckHostIP             no
        ForwardX11              yes
        ForwardAgent            yes
        StrictHostKeyChecking   no
        UsePrivilegedPort       no
        FallBackToRsh           no
        Protocol                1,2
</file>

chmod o+rx /root
mkdir /root/.ssh
chmod o+rx /root/.ssh

</post>
</kickstart>
```

When Things Go Wrong

- Test your Kickstart Graph
  - Check XML syntax: xmllint
  - Make a kickstart file
    - Make kickstart file as a node will see it

```
# rocks list host profile compute-0-0
```

When Things Go Wrong

- Test your Kickstart Graph
  - Check XML syntax: xmllint
    - # cd sweetroll/nodes
    - **# xmllint --noout sweetroll.xml**

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

<kickstart>
  <description>
The sweet roll. This roll is just sweet!
  <description>
</kickstart>
```

```
# xmllint --noout sweetroll.xml

sweetroll.xml:7: parser error : Opening and
ending tag mismatch: description line 6 and
kickstart
</kickstart>
           ^
```

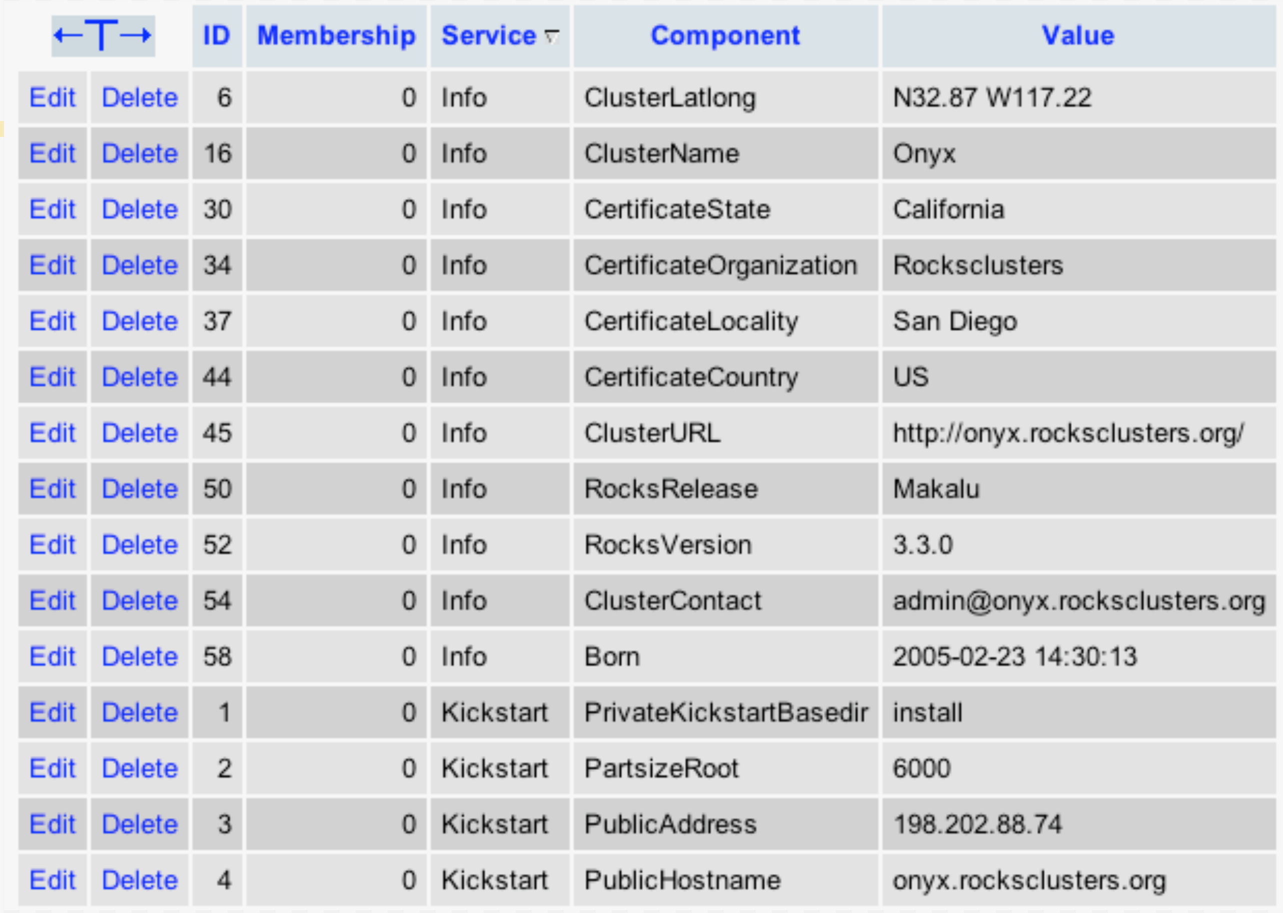
Nodes XML Tools: <var>

- Get Variables from Database
  - `<var name="Kickstart_PrivateAddress"/>`
  - `<var name="Node_Hostname"/>`

```
10.1.1.1
compute-0-0
```

  - Can grab any value from the *app_globals* database table
  - (in 5.2 replaced by Attributes!)

<var> values from app_globals



←T→		ID	Membership	Service ▿	Component	Value
Edit	Delete	6	0	Info	ClusterLatlong	N32.87 W117.22
Edit	Delete	16	0	Info	ClusterName	Onyx
Edit	Delete	30	0	Info	CertificateState	California
Edit	Delete	34	0	Info	CertificateOrganization	Rocksclusters
Edit	Delete	37	0	Info	CertificateLocality	San Diego
Edit	Delete	44	0	Info	CertificateCountry	US
Edit	Delete	45	0	Info	ClusterURL	http://onyx.rocksclusters.org/
Edit	Delete	50	0	Info	RocksRelease	Makalu
Edit	Delete	52	0	Info	RocksVersion	3.3.0
Edit	Delete	54	0	Info	ClusterContact	admin@onyx.rocksclusters.org
Edit	Delete	58	0	Info	Born	2005-02-23 14:30:13
Edit	Delete	1	0	Kickstart	PrivateKickstartBasedir	install
Edit	Delete	2	0	Kickstart	PartsizeRoot	6000
Edit	Delete	3	0	Kickstart	PublicAddress	198.202.88.74
Edit	Delete	4	0	Kickstart	PublicHostname	onyx.rocksclusters.org

- Combine “Service” and “Component”
  - For example, Kickstart_PublicAddress

Adding your own vars

- rocks add var service= component= value=
- Easy place to put variables to reference in your xml files.

Nodes XML Tools: <var>

- <var> attributes
  - name
    - Required. Format is “Service_Component”
    - Service and Component relate to column names in the app_global database table.
  - val
    - Optional. Sets the value of this variable
      - <var name=“Info_ClusterName” val=“Seinfeld”/>
  - ref
    - Optional. Set this variable equal to another
      - <var name=“Info_Weather” ref=“Info_Forecast”/>

Nodes XML Tools: <eval>

- Do processing on the frontend when the kickstart file is generated (by the CGI script):
  - `<eval shell="bash">`
- To insert the Rocks release info in the kickstart file:

```
<eval shell="bash">
cat /etc/rocks-release
</eval>
```

```
Rocks release 4.2.1 (Cydonia)
```

Nodes XML Tools: <eval>

- <eval> attributes
  - shell
    - Optional. The interpreter to use. Default “sh”
  - mode
    - Optional. Value is quote or xml. Default of quote specifies for kpp to escape any XML characters in output.
    - XML mode allows you to generate other tags:
      - <eval shell=“python” mode=“xml”>
        - import time
        - now = time.time()
        - print “<var name=‘Info_Now’ val=‘%s’/>” % now
      - </eval>

Nodes XML Tools: <eval>

- Inside <eval> variables are not accessed with <var>: use the environment instead.

```
<eval shell="python">
import os
print "My NTP time server is",
 os.environ['Kickstart_PublicNTPHost']
print "Got it?"
</eval>
```

```
My NTP time server is time.apple.com
Got it?
```

Nodes XML Tools <file>

- Create a file on the system:
  - `<file name="/etc/hi-mom" mode="append">`
    - `How are you today?`
  - `</file>`
- Used extensively throughout Rocks post sections
  - Keeps track of alterations automatically via RCS.

```
<file name="/etc/hi" perms="444">
How are you today?
I am fine.
</file>
```

```
...RCS checkin commands...
cat > /etc/hi << 'EOF'
How are you today?
I am fine.
EOF
chmod 444 /etc/hi-mom
...RCS cleanup commands...
```

Nodes XML Tools: <file>

- <file> attributes
  - name
    - Required. The full path of the file to write.
  - mode
    - Optional. Value is “create” or “append”. Default is create.
  - owner
    - Optional. Value is “user.group”, can be numbers or names.
      - <file name=“/etc/hi” owner=“daemon.root”>
  - perms
    - Optional. The permissions of the file. Can be any valid “chmod” string.
      - <file name=“/etc/hi” perms=“a+x”>

Nodes XML Tools: <file>

- <file> attributes (continued)
  - vars
    - Optional. Value is “literal” or “expanded”. In literal (default), no variables or backticks in file contents are processed. In expanded, they work normally.
      - <file name=“/etc/hi” vars=“expanded”>
        - The current date is \`date\`
      - </file>
  - expr
    - Optional. Specifies a command (run on the frontend) whose output is placed in the file.
      - <file name=“/etc/hi” expr=“/opt/rocks/dbreport hi”/>

Fancy <file>: nested tags

```
<file name="/etc/hi">

Rocks release:
<eval>
date +"%d-%b-%Y"
echo ""
cat /etc/rocks-release
</eval>

</file>
```

```
…RCS checkin commands...
cat > /etc/hi << 'EOF'

Rocks release:
13-May-2005

Rocks release 4.2.1 (Cydonia)

EOF
…RCS cleanup commands…
```

Look at Rocks Rolls

- Many examples.
- “Copy and edit” is faster than “create and debug”

When it just can be done in the Post

- Some software cannot be configured in the install environment
  - E.g. Condor needs the running env
  - Compiling of specialized add on devices

Two Avenues ---

- /etc/rc.d/rocksconfig.d
- /opt/rocks/SRPMS

Rocks mod to inittab

```
bw::bootwait:/etc/rc.d/rc.rocksconfig before-rc
po:35:wait:/etc/rc.d/rc.rocksconfig after-rc
```

Files like /etc/rc.d/rocksconfig.d/pre-nn-* are excuted before rc.d startup scripts

Files like /etc/rc.d/rocksconfig.d/post-nn-* are executed after rc.d has completed

Taking advantage of rocksconfig.d

- Your roll xml file can lay down an rc/rocksconfig.d file to particular things on boot
- If you only want it done on first boot have the script remove itself after execution.

/opt/rocks/SRPMS

- In the rocksconfig.d/pre-10 script:
  - Any source RPM in /opt/rocks/SRPMS will be rebuilt and installed
  - Useful for device drivers that are not part of kernel (e.g. Myrinet, IB)

Summary

- Look at the Rocks Rolls for examples.
- Rolls are not difficult, Understanding what is going on under the covers helps demystify
- Some software is more challenging than others
- Test. Test. Test.