Bcfg2 Documentation

Release 1.2.0

Narayan Desai et al.

November 09, 2010

CONTENTS

CHAPTER

ONE

INTRODUCTION

Bcfg2 helps system administrators produce a consistent, reproducible, and verifiable description of their environment, and offers visualization and reporting tools to aid in day-to-day administrative tasks. It is the fifth generation of configuration management tools developed in the Mathematics and Computer Science Division of Argonne National Laboratory.

It is based on an operational model in which the specification can be used to validate and optionally change the state of clients, but in a feature unique to Bcfg2 the client's response to the specification can also be used to assess the completeness of the specification. Using this feature, Bcfg2 provides an objective measure of how good a job an administrator has done in specifying the configuration of client systems. Bcfg2 is therefore built to help administrators construct an accurate, comprehensive specification.

Bcfg2 has been designed from the ground up to support gentle reconciliation between the specification and current client states. It is designed to gracefully cope with manual system modifications.

Finally, due to the rapid pace of updates on modern networks, client systems are constantly changing; if required in your environment, Bcfg2 can enable the construction of complex change management and deployment strategies.

1.1 Architecture Overview

Bcfg2 provides a declarative interface to system configuration. Its configuration specifications describe a literal configuration goal state for clients. In this architecture, the Bcfg2 client tool is responsible for determining what, if any, configuration operations must occur and then performing those operations. The client also uploads statistics and client configuration state information. The design and implementation of the reporting system is described on a separate *page*.

A comprehensive description of the Bcfg2 Architecture (and the choices behind the design) can be found at *Architecture in Detail*.

1.1.1 Server

The role of the Bcfg2 server is rendering a client-specific target configuration description from a global specification. The specification consists of a directory structure containing data for a variety of server plugins. The Bcfg2 server has a plugin interface that can be used to interpret the configuration specification.

1.1.2 Client

The Bcfg2 client is responsible for determining what operations are necessary in order to reach the desired configuration state. Read on for more information about *The Bcfg2 Client*.

1.2 What Operating Systems Does Bcfg2 Support?

Bcfg2 is fairly portable. It has been successfully run on:

- AIX, FreeBSD, OpenBSD, Mac OS X, OpenSolaris, Solaris.
- Many GNU/Linux distributions, including Archlinux, Blag, CentOS, Debian, Fedora, Gentoo, gNewSense, Mandriva, OpenSUSE, Red Hat/RHEL, Scientific Linux, SuSE/SLES, Trisquel, and Ubuntu.

Bcfg2 should run on any POSIX compatible operating system, however direct support for an operating system's package and service formats are limited by the currently available *Available client tools* (new client tools are pretty easy to add). Check the *FAQ* for a more exact list of platforms on which Bcfg2 works'.

CHAPTER

TWO

INSTALLATION

Before installing, you will need to choose a machine to be the Bcfg2 server. We recommend a Linux-based machine for this purpose, but the server will work on any supported operating system. Note that you may eventually want to run a web server on this machine for reporting and serving up package repositories. The server package only needs to be installed on your designated Bcfg2 server machine. The clients package needs to be installed on any machine you plan to manage by Bcfg2.

2.1 Prerequisites

Bcfg2 has several server side prerequisites and a minimal set of client side requirements. This page describes the prerequisite software situation on all supported platforms. The table describes what software is needed on the client and server side.

2.1.1 Bcfg2 Client

Software	Version	Requires
libxml2 (if lxml is used)	Any	
libxslt (if lxml is used)	Any	libxml2
python	2.3-2.4, 2.5 [1]	
lxml or elementtree [2]	Any	lxml: libxml2, libxslt, python
python-apt [3]	Any	python
debsums (if APT tool driver is used)	Any	

[1] python 2.5 works with elementtree.
[2] elementtree is included in python 2.5 and later.
[3] python-apt is only required on platforms that use apt, such as Debian and Ubuntu.

2.1.2 Bcfg2 Server

Software	Version	Requires
libxml2	2.6.24+	
libxslt	Any	libxml2
python	2.2-2.5	
lxml	0.9+	lxml: libxml2, libxslt, python
gamin or fam	Any	
python-gamin or python-fam	Any	gamin or fam, python
M2crypto	Any	python, openssl

2.2 Download

2.2.1 Tarball

The Bcfg2 source tarball can be grabbed from the Download page.

Version	URL	GPG key	Release Date	md5sum
1.1.0	Tarball	GPGKey	9/27/10	13593938daf7e8b9a81cb4b677dc7f99

The full command to use with `wget` are listed below. Please replace `<version>` with 1.1.0.

```
wget ftp://ftp.mcs.anl.gov/pub/bcfg/bcfg2-<version>.tar.gz
wget ftp://ftp.mcs.anl.gov/pub/bcfg/bcfg2-<version>.tar.gz.gpg
```

All tarballs are signed with GPG key 7F7D197E. You can verify your download by importing the key and running

```
$ gpg --recv-keys 0x75bf2c177f7d197e
$ gpg --verify bcfg2-<version>.tar.gz.gpg bcfg2-<version>.tar.gz
```

For older or prepreleases please visit the Download wiki page.

2.2.2 Git checkout

You can also get the latest (possibly broken) code via git

```
git clone git://git.mcs.anl.gov/bcfg2.git
```

2.3 Installation from source

If you are working with the release tarball of Bcfg2 you need to untar it before you can go on with the installation

```
tar -xzf bcfg2-<version>.tar.gz
```

Now you can build Bcfg2 with. If you are working with a SVN checkout no <version> need to be specified.

```
cd bcfg2-<version>
python setup.py install --prefix=/install/prefix
```

This will install both the client and server on that machine.

2.4 Building packages from source

The Bcfg2 distribution contains two different spec files.

2.4.1 Building from Tarball

- Copy the tarball to */usr/src/packages/SOURCES/*
- Extract another copy of it somewhere else (eg: */tmp*) and retrieve the *misc/bcfg2.spec* file
- Run

  ```
  rpmbuild -ba bcfg2.spec
  ```

- The resulting RPMs will be in */usr/src/packages/RPMS/* and SRPMs in */usr/src/packages/SRPMS*

2.4.2 Building from an GIT Checkout

- Change to the *redhat/* directory in the working copy
- Run

  ```
  make
  ```

- The resulting RPMs will be in */usr/src/redhat/RPMS/ 'and SRPMs in '/usr/src/redhat/SRPMS* and will have the SVN revision appended

2.4.3 Building RPM packages with `rpmbuild`

While you can go about building all these things from source, this how to will try and meet the dependencies using packages from EPEL. The *el5* package should be compatible with CentOS 5.x.

- Installation of the EPEL repository package

  ```
  [root@centos ~]# rpm -Uvh http://download.fedora.redhat.com/pub/epel/5/i386/epel-rel
  ```

- Now you can install the rest of the prerequisites

  ```
  [root@centos ~]# yum install python-genshi python-cheetah python-lxml
  ```

- After installing git, check out the master branch

  ```
  [root@centos redhat]# git clone git://git.mcs.anl.gov/bcfg2.git
  ```

- Install the `fedora-packager` package

```
[root@centos ~]# yum install fedora-packager
```

- A directory structure for the RPM build process has to be established.

```
[you@centos ~]$ rpmdev-setuptree
```

- Change to the *redhat* directory of the checked out Bcfg2 source:

```
[you@centos ~]$ cd bcfg2/redhat/
```

- In the particular directory is a `Makefile` which will do the job of building the RPM packages. You can do this as root, but it's not recommanded

```
[you@centos redhat]$ make
```

- Now the new RPM package can be installed. Please adjust the path to your RPM package

```
[root@centos ~]# rpm -ihv /home/YOU/rpmbuild/RPMS/noarch/bcfg2-server-1.0.0-0.2r5835
```

2.5 Distribution-specific notes

The installation of Bcfg2 on a specific distribution depends on the used package management tool and the disposability in the distribution's package *repository* .

2.5.1 ArchLinux

Packages for Arch Linux are available in the Arch User Repository (AUR). Just use *pacman* to perform the installation

```
pacman -S bcfg2 bcfg2-server
```

2.5.2 Debian

Packages of Bcfg2 are available for Debian Lenny, Debian Squeeze, and Debian Sid. The fastest way to get Bcfg2 onto your Debian system is to use *apt-get* or *aptitude*.

```
sudo aptitude install bcfg2 bcfg2-server
```

If you want to use unofficial packages from Bcfg2. Add the following line to your */etc/apt/sources.list* file

```
deb ftp://ftp.mcs.anl.gov/pub/bcfg/debian sarge/
```

Now just run *aptitute* in the way it is mentioned above.

For more details about running prerelease version of Bcfg2 on Debian systems, please refer to the Wiki.

2.5.3 Fedora

The fastest way to get Bcfg2 Packages onto your Fedora system is to use *yum* or PackageKit. Yum will pull in all dependencies of Bcfg2 automatically.

```
$ su -c 'yum install bcfg2-server bcfg2'
```

Be aware that the latest release of Bcfg2 may only be available for the Development release of Fedora (Rawhide). With the activation of the Rawhide repository of Fedora you will be able to install it.

```
$ su -c 'yum install --enablerepo=rawhide bcfg2-server bcfg2'
```

This way is not recommanded on productive systems. Only for testing.

2.5.4 Gentoo

Early in July 2008, Bcfg2 was added to the Gentoo portage tree. So far it's only keyworded for ~x86, but we hope to see it soon in the amd64 and x64-solaris ports. If you're using Gentoo on some other architecture, it should still work provided that you have a reasonably up to date Python; try adding *app-admin/bcfg2 ~** to your */etc/portage/package.keywords* file.

If you don't use portage to install Bcfg2, you'll want to make sure you have all the prerequisites installed first. For a server, you'll need:

- `app-admin/gamin` or `app-admin/fam`
- `dev-python/lxml`

Clients will need at least:

- `app-portage/gentoolkit`

2.5.5 OS X

Once macports is installed:

```
port install bcfg2
```

Using native OS X python

First, make sure you have Xcode installed as you need *packagemaker* which comes bundled in the Developer tools.

Clone the git source:

```
git clone git://git.mcs.anl.gov/bcfg2.git
```

Change to the osx directory and type make. Your new package should be located at bcfg2-‘‘‘$VERSION’‘’.pkg (where ‘‘‘$VERSION’‘’ is that which is specified in setup.py).

2.5.6 RHEL / Centos / Scientific Linux

While you can go about building all these things from source, this section will try and meet the dependencies using packages from EPEL [4]. The *el5* and the soon available *el6* package should be compatible with CentOS 5.x/6.x and Scientific Linux.

EPEL:

```
[root@centos ~]# rpm -Uvh http://download.fedora.redhat.com/pub/epel/5/i386/epel-release-5-
```

Install the bcfg2-server and bcfg2 RPMs:

```
[root@centos ~]# yum install bcfg2-server bcfg2
```

Note: The latest package for *el5* is only available in the testing repository.

2.5.7 Ubuntu

The steps to bring Bcfg2 onto your Ubuntu system depends on your release.

- **Dapper** Add the following lines to */etc/apt/sources.list*

```
deb ftp://ftp.mcs.anl.gov/pub/bcfg/ubuntu dapper/
deb http://archive.ubuntu.com/ubuntu dapper universe
deb-src http://archive.ubuntu.com/ubuntu dapper universe
```

- **Edgy** Add the following lines to */etc/apt/sources.list*

```
deb ftp://ftp.mcs.anl.gov/pub/bcfg/ubuntu edgy/
deb http://archive.ubuntu.com/ubuntu edgy universe
deb-src http://archive.ubuntu.com/ubuntu edgy universe
```

- **Feisty** Those packages are available from the Ubuntu repositories.

To install the packages, just lauch the following command

```
sudo aptitude install bcfg2 bcfg2-server
```

For more details about running prerelease version of Bcfg2 on Ubuntu systems, please refer to the Wiki.

[4] For more details check the EPEL instructions

CHAPTER

THREE

GETTING STARTED

The steps below should get you from just thinking about a configuration management system to an operational installation of Bcfg2. If you get stuck, be sure to check the *mailinglist* or to drop in on our *IRC Channel*.

3.1 Get and Install Bcfg2 Server

We recommend running the server on a Linux machine for ease of deployment due to the availability of packages for the dependencies.

First, you need to download and install Bcfg2. The section *Installation* in this manual describes the steps to take. To start, you will need to install the server on one machine and the client on one or more machines. Yes, your server can also be a client (and should be by the time your environment is fully managed).

3.2 Set up Repository

The next step after installing the Bcfg2 packages is to configure the server. You can easily set up a personalized default configuration by running, on the server,

```
bcfg2-admin init
```

You will be presented with a series of questions that will build a Bcfg2 configuration file in `/etc/bcfg2.conf`, set up a skeleton repository (in `/var/lib/bcfg2` by default), help you create ssl certificates, and do any other similar tasks needed to get you started.

Once this process is done, you can start the Bcfg2 server:

```
/etc/init.d/bcfg2-server start
```

You can try it out by running the Bcfg2 client on the same machine, acting like it is your first client.

Note: The following command will tell the client to run in no-op mode, meaning it will only check the client against the repository and report any changes it sees. It won't make any changes (partially because you haven't populated the repository with any yet). However, nobody is perfect - you can make a typo, our software can have bugs, monkeys can break in and hit enter before you are done. Don't run this command

on a production system if you don't know what it does and aren't prepared for the consequences. We don't know of anybody having problems with it before, but it is better to be safe than sorry.

And now for the command:

```
bcfg2 -q -v -n
```

That can be translated as "bcfg2 quick verbose no-op". The output should be something similar to:

```
Loaded tool drivers:
 Chkconfig    POSIX        PostInstall  RPM

Phase: initial
Correct entries:        0
Incorrect entries:      0
Total managed entries:  0
Unmanaged entries:      242


Phase: final
Correct entries:        0
Incorrect entries:      0
Total managed entries:  0
Unmanaged entries:      242
```

Perfect! We have started out with an empty configuration, and none of our configuration elements are correct. It doesn't get much cleaner than that. But what about those unmanaged entries? Those are the extra configuration elements (probably all packages at the moment) that still aren't managed. Your goal now is to migrate each of those plus any it can't see up to the "Correct entries" line.

3.3 Populate Repository

Finally, you need to populate your repository. Unfortunately, from here on out we can't write up a simple recipe for you to follow to get this done. It is very dependent on your local configuration, your configuration management goals, the politics surrounding your particular machines, and many other similar details. We can, however, give you guidance.

After the above steps, you should have a toplevel repository structure that looks like:

```
bcfg-server:~ # ls /var/lib/bcfg2
Bundler/  Cfg/  Metadata/  Pkgmgr/  Rules/  SSHbase/  etc/
```

The place to start is the Metadata directory, which contains two files: `clients.xml` and `groups.xml`. Your current `clients.xml` will look pretty close to:

```
<Clients version="3.0">
   <Client profile="basic" pingable="Y" pingtime="0" name="bcfg-server.example.com"/>
</Clients>
```

The `clients.xml` file is just a series of `<Client />` tags, each of which describe one host you manage. Right now we only manage one host, the server machine we just created. This machine is bound to the `basic` profile, is pingable, has a pingtime of `0`, and has the name `bcfg-server.example.com`. The

two "ping" parameters don't matter to us at the moment, but the other two do. The name parameter is the fully qualified domain name of your host, and the profile parameter maps that host into the `groups.xml` file.

Our simple `groups.xml` file looks like:

```
<Groups version='3.0'>
   <Group profile='true' public='false' name='basic'>
      <Group name='suse'/>
   </Group>
   <Group name='ubuntu' toolset='debian'/>
   <Group name='debian' toolset='debian'/>
   <Group name='redhat' toolset='rh'/>
   <Group name='suse' toolset='rh'/>
   <Group name='mandrake' toolset='rh'/>
   <Group name='solaris' toolset='solaris'/>
</Groups>
```

There are two types of groups in Bcfg: profile groups (`profile='true'`) and non-profile groups (`profile='false'`). Profile groups can act as top-level groups to which clients can bind, while non-profile groups only exist as members of other groups. In our simple starter case, we have a profile group named `basic`, and that is the group that our first client bound to. Our first client is a SuSE machine, so it contains the `suse` group. Of course, `bcfg2-admin` isn't smart enough to fill out the rest of your config, so the `suse` group further down is empty.

Let's say the first thing we want to set up on our machine is the message of the day. To do this, we simply need to create a Bundle and add that Bundle to an appropriate group. In this simple example, we start out by adding

```
<Bundle name='motd'/>
```

to the `basic` group.

Next, we create a motd.xml file in the Bundler directory:

```
<Bundle name='motd' version='2.0'>
  <Path name='/etc/motd' />
</Bundle>
```

Now when we run the client, we get slightly different output:

```
Loaded tool drivers:
 Chkconfig    POSIX        PostInstall  RPM
Incomplete information for entry Path:/etc/motd; cannot verify

Phase: initial
Correct entries:        0
Incorrect entries:      1
Total managed entries:  1
Unmanaged entries:      242

In dryrun mode: suppressing entry installation for:
 Path:/etc/motd
```

```
Phase: final
Correct entries:        0
Incorrect entries:      1
Total managed entries:  1
Unmanaged entries:      242
```

We now have an extra unmanaged entry, bringing our total number of managed entries up to one. To manage it we need to copy `/etc/motd` to `/var/lib/bcfg2/Cfg/etc/motd/`. Note the layout of that path: all plain-text config files live in the Cfg directory. The directory structure under that directory directly mimics your real filesystem layout, making it easy to find and add new files. The last directory is the name of the file itself, so in this case the full path to the motd file would be `/var/lib/bcfg2/Cfg/etc/motd/motd`. Copy your real `/etc/motd` file to that location, run the client again, and you will find that we now have a correct entry:

```
Loaded tool drivers:
 Chkconfig    POSIX        PostInstall  RPM

Phase: initial
Correct entries:        1
Incorrect entries:      0
Total managed entries:  1
Unmanaged entries:      242


Phase: final
Correct entries:        1
Incorrect entries:      0
Total managed entries:  1
Unmanaged entries:      242
```

Done! Now we just have 242 (or more) entries to take care of!

Bundler is a relatively easy directory to populate. You can find many samples of Bundles in the Bundle Repository, many of which can be used without editing.

3.4 Next Steps

Several other utilities can help from this point on:

`bcfg2-info` is a utility that instantiates a copy of the Bcfg2 server core (minus the networking code) for examination. From this, you can directly query:

- Client Metadata
- Which entries are provided by particular plugins
- Client Configurations

Run `bcfg2-info`, and type help and the prompt when it comes up.

`bcfg2-admin` can perform a variety of repository maintenance tasks. Run `bcfg2-admin` help for details.

CHAPTER

FOUR

ARCHITECTURE IN DETAIL

Bcfg2 is based on a client-server architecture. The client is responsible for interpreting (but not processing) the configuration served by the server. This configuration is literal, so no local process is required. After completion of the configuration process, the client uploads a set of statistics to the server. This section will describe the goals and then the architecture motivated by it.

4.1 Goals

- **Model configurations using declarative semantics.**

  Declarative semantics maximize the utility of configuration management tools; they provide the most flexibility for the tool to determine the right course of action in any given situation. This means that users can focus on the task of describing the desired configuration, while leaving the task of transitioning clients states to the tool.

- **Configuration descriptions should be comprehensive.**

  This means that configurations served to the client should be sufficient to reproduce all desired functionality. This assumption allows the use of heuristics to detect extra configuration, aiding in reliable, comprehensive configuration definitions.

- **Provide a flexible approach to user interactions.**

  Most configuration management systems take a rigid approach to user interactions; that is, either the client system is always correct, or the central system is. This means that users are forced into an overly proscribed model where the system asserts where correct data is. Configuration data modification is frequently undertaken on both the configuration server and clients. Hence, the existence of a single canonical data location can easily pose a problem during normal tool use. Bcfg2 takes a different approach.

The default assumption is that data on the server is correct, however, the client has the option to run in another mode where local changes are catalogued for server-side integration. If the Bcfg2 client is run in dry run mode, it can help to reconcile differences between current client state and the configuration described on the server. The Bcfg2 client also searches for extra configuration; that is, configuration that is not specified by the configuration description. When extra configuration is found, either configuration has been removed from the configuration description on the server, or manual configuration has occurred on the

client. Options related to two-way verification and removal are useful for configuration reconciliation when interactive access is used.

- Plugins and administrative applications.
- Incremental operations.

4.2 The Bcfg2 Client

The Bcfg2 client performs all client configuration or reconfiguration operations. It renders a declarative configuration specification, provided by the Bcfg2 server, into a set of configuration operations which will, if executed, attempt to change the client's state into that described by the configuration specification. Conceptually, the Bcfg2 client serves to isolate the Bcfg2 server and specification from the imperative operations required to implement configuration changes.

This isolation allows declarative specifications to be manipulated symbolically on the server, without needing to understand the properties of the underlying system tools. In this way, the Bcfg2 client acts as a sort of expert system that *knows* how to implement declarative configuration changes.

The operation of the Bcfg2 client is intended to be as simple as possible. The normal configuration process consists of four main steps:

- **Probe Execution**

  During the probe execution stage, the client connects to the server and downloads a series of probes to execute. These probes reveal local facts to the Bcfg2 server. For example, a probe could discover the type of video card in a system. The Bcfg2 client returns this data to the server, where it can influence the client configuration generation process.

- **Configuration Download and Inventory**

  The Bcfg2 client now downloads a configuration specification from the Bcfg2 server. The configuration describes the complete target state of the machine. That is, all aspects of client configuration should be represented in this specification. For example, all software packages and services should be represented in the configuration specification. The client now performs a local system inventory. This process consists of verifying each entry present in the configuration specification. After this check is completed, heuristic checks are executed for configuration not included in the configuration specification. We refer to this inventory process as 2-way validation, as first we verify that the client contains all configuration that is included in the specification, then we check if the client has any extra configuration that isn't present. This provides a fairly rigorous notion of client configuration congruence. Once the 2-way verification process has been performed, the client has built a list of all configuration entries that are out of spec. This list has two parts: specified configuration that is incorrect (or missing) and unspecified configuration that should be removed.

- **Configuration Update**

  The client now attempts to update its configuration to match the specification. Depending on options, changes may not (or only partially) be performed. First, if extra configuration correction is enabled, extra configuration can be removed. Then the remaining changes are processed. The Bcfg2 client loops while progress is made in the correction of these incorrect configuration entries. This loop results in the client being able to accomplish all it will be able to during one execution. Once all

entries are fixed, or no progress is being made, the loop terminates. Once all configuration changes that can be performed have been, bundle dependencies are handled. Bundle groupings result in two different behaviors. Contained entries are assumed to be inter-dependent. To address this, the client re-verifies each entry in any bundle containing an updates configuration entry. Also, services contained in modified bundles are restarted.

- **Statistics Upload**

  Once the reconfiguration process has concluded, the client reports information back to the server about the actions it performed during the reconfiguration process. Statistics function as a detailed return code from the client. The server stores statistics information. Information included in this statistics update includes (but is not limited to):

  - Overall client status (clean/dirty)
  - List of modified configuration entries
  - List of uncorrectable configuration entries
  - List of unmanaged configuration entries

4.2.1 Architecture Abstraction

The Bcfg2 client internally supports the administrative tools available on different architectures. For example, `rpm` and `apt-get` are both supported, allowing operation of Debian, Redhat, SUSE, and Mandriva systems. The client toolset is determined based on the availability of client tools. The client includes a series of libraries which describe how to interact with the system tools on a particular platform.

Three of the libraries exist. There is a base set of functions, which contain definitions describing how to perform POSIX operations. Support for configuration files, directories, symlinks, hardlinks, etc., are included here. Two other libraries subclass this one, providing support for Debian and rpm-based systems.

The Debian toolset includes support for apt-get and update-rc.d. These tools provide the ability to install and remove packages, and to install and remove services.

The Redhat toolset includes support for rpm and chkconfig. Any other platform that uses these tools can also use this toolset. Hence, all of the other familiar rpm-based distributions can use this toolset without issue.

Other platforms can easily use the POSIX toolset, ignoring support for packages or services. Alternatively, adding support for new toolsets isn't difficult. Each toolset consists of about 125 lines of python code.

4.3 The Bcfg2 Server

The Bcfg2 server is responsible for taking a network description and turning it into a series of configuration specifications for particular clients. It also manages probed data and tracks statistics for clients.

The Bcfg2 server takes information from two sources when generating client configuration specifications. The first is a pool of metadata that describes clients as members of an aspect-based classing system. That

is, clients are defined in terms of aspects of their behavior. The other is a file system repository that contains mappings from metadata to literal configuration. These are combined to form the literal configuration specifications for clients.

4.3.1 The Configuration Specification Construction Process

As we described in the previous section, the client connects to the server to request a configuration specification. The server uses the client's metadata and the file system repository to build a specification that is tailored for the client. This process consists of the following steps:

- **Metadata Lookup**

  The server uses the client's IP address to initiate the metadata lookup. This initial metadata consists of a (profile, image) tuple. If the client already has metadata registered, then it is used. If not, then default values are used and stored for future use. This metadata tuple is expanded using some profile and class definitions also included in the metadata. The end result of this process is metadata consisting of hostname, profile, image, a list of classes, a list of attributes and a list of bundles.

- **Abstract Configuration Construction**

  Once the server has the client metadata, it is used to create an abstract configuration. An abstract configuration contains all of the configuration elements that will exist in the final specification **without** any specifics. All entries will be typed (i.e. the tagname will be one of Package, Path, Action, etc) and will include a name. These configuration entries are grouped into bundles, which document installation time interdependencies.

- **Configuration Binding**

  The abstract configuration determines the structure of the client configuration, however, it doesn't yet contain literal configuration information. After the abstract configuration is created, each configuration entry must be bound to a client-specific value. The Bcfg2 server uses plugins to provide these client-specific bindings. The Bcfg2 server core contains a dispatch table that describes which plugins can handle requests of a particular type. The responsible plugin is located for each entry. It is called, passing in the configuration entry and the client's metadata. The behavior of plugins is explicitly undefined, so as to allow maximum flexibility. The behaviours of the stock plugins are documented elsewhere in this manual. Once this binding process is completed, the server has a literal, client-specific configuration specification. This specification is complete and comprehensive; the client doesn't need to process it at all in order to use it. It also represents the totality of the configuration specified for the client.

4.4 The Literal Configuration Specification

Literal configuration specifications are served to clients by the Bcfg2 server. This is a differentiating factor for Bcfg2; all other major configuration management systems use a non-literal configuration specification. That is, the clients receive a symbolic configuration that they process to implement target states. We took the literal approach for a few reasons:

- A small list of configuration element types can be defined, each of which can have a set of defined semantics. This allows the server to have a well-formed model of client-side operations. Without a

static lexicon with defined semantics, this isn't possible. This allows the server, for example, to record the update of a package as a coherent event.

- Literal configurations do not require client-side processing. Removing client-side processing reduces the critical footprint of the tool. That is, the Bcfg2 client (and the tools it calls) need to be functional, but the rest of the system can be in any state. Yet, the client will receive a correct configuration.
- Having static, defined element semantics also requires that all operations be defined and implemented in advance. The implementation can maximize reliability and robustness. In more ad-hoc setups, these operations aren't necessarily safely implemented.

4.4.1 The Structure of Specifications

Configuration specifications contain some number of clauses. Two types of clauses exist. Bundles are groups of inter-dependent configuration entities. The purpose of bundles is to encode installation-time dependencies such that all new configuration is properly activated during reconfiguration operations. That is, if a daemon configuration file is changed, its daemon should be restarted. Another example of bundle usage is the reconfiguration of a software package. If a package contains a default configuration file, but it gets overwritten by an environment-specific one, then that updated configuration file should survive package upgrade. The purpose of bundles is to describe services, or reconfigured software packages. Independent clauses contain groups of configuration entities that aren't related in any way. This provides a convenient mechanism that can be used for bulk installations of software.

Each of these clauses contains some number of configuration entities. A number of configuration entities exist including Path, Package, Service, etc. Each of these correspond to the obvious system item. Configuration specifications can get quite large; many systems have specifications that top one megabyte in size. An example of one is included in an appendix. These configurations can be written by hand, or generated by the server.

4.5 Design Considerations

This section will discuss several aspects of the design of Bcfg2, and the particular use cases that motivated them. Initially, this will consist of a discussion of the system metadata, and the intended usage model for package indices as well.

4.5.1 System Metadata

Bcfg2 system metadata describes the underlying patterns in system configurations. It describes commonalities and differences between these specifications in a rigorous way. The groups used by Bcfg2's metadata are responsible for differentiating clients from one another, and building collections of allocatable configuration.

The Bcfg2 metadata system has been designed with several high-level goals in mind. Flexibility and precision are paramount concerns; no configuration should be undescribable using the constructs present in the Bcfg2 repository. We have found (generally the hard way) that any assumptions about the inherent simplicity of configuration patterns tend to be wrong, so obscenely complex configurations must be representable, even if these requirements seem illogical during the implementation.

In particular, we wanted to streamline several operations that commonly occurred in our environment.

- Copying one node's profile to another node.

  In many environments, many nodes are instances of a common configuration specification. They all have similar roles and software. In our environment, desktop machines were the best example of this. Other than strictly per-host configuration like SSH keys, all desktop machines use a common configuration specification. This trivializes the process of creating a new desktop machine.

- Creating a specialized version of an existing profile.

  In environments with highly varied configurations, departmental infrastructure being a good example, "another machine like X but with extra software" is a common requirement. For this reason, it must be trivially possible to inherit most of a configuration specification from some more generic source, while being able to describe overriding aspects in a convenient fashion.

- Compose several pre-existing configuration aspects to create a new profile.

  The ability to compose configuration aspects allows the easy creation of new profiles based on a series of predefined set of configuration specification fragments. The end result is more agility in environments where change is the norm.

  In order for a classing system to be comprehensive, it must be usable in complex ways. The Bcfg2 metadata system has constructs that map cleanly to first-order logic. This implies that any complex configuration pattern can be represented (at all) by the metadata, as first-order logic is provably comprehensive. (There is a discussion later in the document describing the metadata system in detail, and showing how it corresponds to first-order logic)

These use cases motivate several of the design decisions that we made. There must be a many to one correspondence between clients and groups. Membership in a given profile group must imbue a client with all of its configuration properties.

4.5.2 Package Management

The interface provided in the Bcfg2 repository for package specification was designed with automation in mind. The goal was to support an append only interface to the repository, so that users do not need to continuously re-write already existing bits of specification.

CHAPTER

FIVE

THE BCFG2 SERVER

5.1 Plugins

Plugins are the source of all logic used in building a config. They can perform one of several tasks:

1. Generating configuration inventory lists for clients
2. Generating configuration entry contents for clients
3. Probing client-side state (like hardware inventory, etc) – the generic client probing mechanism is described at *Probes*.
4. Automating administrative tasks (e.g. *SSHbase* which automates ssh key management)
5. Generating client per-entry installation decision-lists

5.1.1 Enabling Plugins

In order for the Bcfg2 server to use a plugin, it needs to be listed on the *plugins* line in `bcfg2.conf`.

5.1.2 Default Plugins

The Bcfg2 repository has the default plugin list currently distributed with Bcfg2: http://trac.mcs.anl.gov/projects/bcfg2/browser/trunk/bcfg2/src/lib/Server/Plugins.

Connectors

Properties

The Properties plugin is a connector plugin that adds information from properties files into client metadata instances.

Enabling Properties First, `mkdir /var/lib/bcfg2/Properties`. Each property XML file goes in this directory. Each will automatically be cached by the server, and reread/reparsed upon changes. Add **Properties** to your *plugins* line in */etc/bcfg2.conf*.

Data Structures Properties adds a new dictionary to client metadata instances that maps property file names to PropertyFile instances. PropertyFile instances contain parsed XML data as the "data" attribute.

Usage Specific property files can be referred to in templates as metadata.Properties[<filename>]. The data attribute is an LXML element object. (Documented here)

Currently, no access methods are defined for this data, but as we formulate common use cases, we will add them to the !PropertyFile class as methods. This will simplify templates.

Accessing Properties contest from TGenshi Access contents of *Properties/auth.xml*

```
${metadata.Properties['auth.xml'].data.find('file').find('bcfg2.key').text}
```

Each of these plugins has a corresponding subdirectory with the same name in the Bcfg2 repository.

Metadata (Grouping)

GroupPatterns

The GroupPatterns plugin is a connector that can assign clients group membership pased on patterns in client hostnames. Two basic methods are supported: regular expressions (NamePatterns) and ranges (NameRange). Hosts that match the specification are placed in the group or groups specified by the pattern.

Setup

1. Enable the GroupPatterns plugin
2. Create the GroupPatterns/config.xml file (similar to the example below).
3. Client groups will be augmented based on the specification

Pattern Types NamePatterns use regular expressions to match client hostnames. All matching clients are placed in the resulting groups. NamePatterns also have the ability to use regular expression matched groups to dynamically create group names. The first two examples below are NamePatterns. The first adds client hostname to both groups gp-test1 and gp-test2. The second matches the hostname as a group and places the client in a group called group-<hostname>.

NameRange patterns allow the use of the application of numeric ranges to host names. The final pattern below matches any of node1-node32 and places them all into the rack1 group. Dynamically generated group names are not supported with NameRange.

Examples

```
<GroupPatterns>
  <GroupPattern>
    <NamePattern>hostname</NamePattern>
    <Group>gp-test1</Group>
    <Group>gp-test2</Group>
```

```
  </GroupPattern>
  <GroupPattern>
    <NamePattern>(.*)</NamePattern>
    <Group>group-$1'</Group>
  </GroupPattern>
  <GroupPattern>
    <NameRange>node[[1-32]]</NameRange>
    <Group>rack1</Group>
  </GroupPattern>
</GroupPatterns>
```

Cluster Example Functional aspects are extracted from hostname strings, and dynamic groups are created.

Expected hostname to group mapping:

```
xnfs1.example.com     -> nfs-server
xnfs2.example.com     -> nfs-server
xlogin1.example.com   -> login-server
xlogin2.example.com   -> login-server
xpvfs1.example.com    -> pvfs-server
xpvfs2.example.com    -> pvfs-server
xwww.example.com      -> www-server
```

GroupPatterns configuration:

```
<GroupPatterns>
  <GroupPattern>
    <NamePattern>^x(\w[^\d|\.]+)\d*\..*</NamePattern>
    <Group>$1-server</Group>
  </GroupPattern>
</GroupPatterns>
```

Regex explanation:

1. !^x Match any hostname that begins with “x”
2. (w[!^d|.]+) followed by one or more word characters that are not a decimal digit or ”.” and save the string to $1
3. d* followed by 0 or more decimal digit(s)
4. ..* followed by a ”.”
5. .* followed by 1 or more of anything else.

Metadata

The metadata mechanism has two types of information, client metadata and group metadata. The client metadata describes which top level group a client is associated with.The group metadata describes groups in terms of what bundles and other groups they include. Each aspect grouping and clients’ memberships are reflected in the Metadata/groups.xml and Metadata/clients.xml files, respectively.

Usage of Groups in Metadata Clients are assigned membership of groups in the Metadata descriptions. Clients can be directly assigned to 'profile' or 'public' groups. Client membership of all other groups is by those groups being associated with the profile or public groups. This file can be indirectly modified from clients through use of the -p flag to bcfg2.

Clients are associated with profile groups in Metadata/clients.xml as shown below.

Metadata/clients.xml The Metadata/clients.xml file contains the mappings of Profile Groups to clients. The file is just a series of <Client /> tags, each of which describe one host. A sample file is below:

```
<Clients version="3.0">
  <Client profile="backup-server" pingable="Y" pingtime="0" name="backup.example.com"/>
  <Client profile="console-server" pingable="Y" pingtime="0" name="con.example.com"/>
  <Client profile="kerberos-master" pingable="Y" pingtime="0" name="kdc.example.com"/>
  <Client profile="mail-server" pingable="Y" pingtime="0" name="mail.example.com"/>
  <Client name='foo' address='10.0.0.1' pingable='N' pingtime='-1'>
      <Alias name='foo-mgmt' address='10.1.0.1'/>
  </Client>
</Clients>
```

Clients Tag The Clients tag has the following possible attributes:

Name	Description	Values
version	Client schema version	String

Client Tag Each entry in clients.xml **must** have the following properties:

Name	Description	Values
name	Host name of client. This needs to be the name (possibly a FQDN) returned by a reverse lookup on the connecting IP address.	String
profile	Profile group name to associate this client with.	String

Additionally, the following properties can be specified:

Name	Description	Values
Alias	Alternative name and address for the client.	XML Element
address	Establishes an extra IP address that resolves to this client.	ip address
location	Requires requests to come from an IP address that matches the client record.	fixed|floating
password	Establishes a per-node password that can be used instead of the global password.	String
pingable	If the client is pingable (deprecated; for old reporting system)	Y|N
pingtime	Last time the client was pingable (deprecated; for old reporting system)	String
secure	Requires the use of the per-client password for this client.	true|false
uuid	Establishes a per-node name that can be used to bypass dns-based client resolution.	String

For detailed information on client authentication see *authentication*

Metadata/groups.xml The `Metadata/groups.xml` file contains Group and Profile definitions. Here's a simple `Metadata/groups.xml` file:

```
<Groups version='3.0'>
  <Group name='mail-server' profile='true'
                           public='false'
                           comment='Top level mail server group' >
    <Bundle name='mail-server'/>
    <Bundle name='mailman-server'/>
    <Group name='apache-server'/>
    <Group name='rhel-as-4-x86'/>
    <Group name='nfs-client'/>
    <Group name='server'/>
  </Group>
  <Group name='rhel-as-4-x86'>
     <Group name='rhel'/>
  </Group>
  <Group name='apache-server'/>
  <Group name='nfs-client'/>
  <Group name='server'/>
  <Group name='rhel'/>
</Groups>
```

Nested/chained groups definitions are conjunctive (logical and). For instance, in the above example, a client associated with the Profile Group `mail-server` is also a member of the `apache-server`, `rhel-as-4-x86`, `nfs-client`, `server` and `rhel` groups.

Groups describe clients in terms for abstract, disjoint aspects. Groups can be combined to form complex descriptions of clients that use configuration commonality and inheritance. Groups have several attributes, described below:

Metadata Groups Tag The Groups tag has the following possible attributes:

Name	Description	Values
version	Group schema version	String
origin	URL of master version (for common repository)	String
revision	Master version control revision	String

Metadata Group Tag The Group Tag has the following possible attributes:

Name	Description	Values
name	Name of the group	String
profile	If a client can be directly associated with this group	True|False
public	If a client can freely associate itself with this group. For use with the *bcfg2 -p* option on the client.	True|False
category	A group can only contain one instance of a group in any one category. This provides the basis for representing groups which are conjugates of one another in a rigorous way. It also provides the basis for negation.	String
default	Set as the profile to use for clients that are not associated with a profile in `clients.xml`	True|False
comment	English text description of group	String

Groups can also contain other groups and bundles.

Use of XInclude XInclude is a W3C specification for the inclusion of external XML documents into XML source files. Much like the use of `#include` in C, this allows complex definitions to be split into smaller, more manageable pieces. As of bcfg2-0.9.0pre1, the Metadata plugin supports the use of XInclude specifications to split the `clients.xml` and `groups.xml` files. This mechanism allows the following specification to produce useful results:

```
<Groups version='3.0' xmlns:xi="http://www.w3.org/2001/XInclude">
 <xi:include href="my-groups.xml" />
 <xi:include href="their-groups.xml" />
</Groups>
```

Each of the included groups files has the same format. These files are properly validated by `bcfg2-repo-validate`. This mechanism is useful for composing group definitions from multiple sources, or setting different permissions in an svn repository.

Probes The metadata plugin includes client-side probing functionality. This is fully documented *here*.

Each of these plugins has a corresponding subdirectory with the same name in the Bcfg2 repository.

Abstract Configuration (Structures)

Bundler

Bundler is used to describe groups of inter-dependent configuration entries, such as the combination of packages, configuration files, and service activations that comprise typical Unix daemons. Bundles are used to add groups of configuration entries to the inventory of client configurations, as opposed to describing particular versions of those entries. For example, a bundle could say that the configuration file `/etc/passwd` should be included in a configuration, but will not describe the particular version of `/etc/passwd` that a given client will receive.

Groups can be used inside of bundles to differentiate which entries particular clients will recieve; this is useful for the case where entries are named differently across systems; for example, one linux distro may have a package called openssh while another uses the name ssh. Configuration entries nested inside of Group elements only apply to clients who are a member of those groups; multiple nested groups must all apply. Also, groups may be negated; entries included in such groups will only apply to clients who are not a member of said group.

The following is an annotated copy of a bundle:

```
<Bundle name='ssh' version='2.0'>
   <Path name='/etc/ssh/ssh_host_dsa_key'/>
   <Path name='/etc/ssh/ssh_host_rsa_key'/>
   <Path name='/etc/ssh/ssh_host_dsa_key.pub'/>
   <Path name='/etc/ssh/ssh_host_rsa_key.pub'/>
   <Path name='/etc/ssh/ssh_host_key'/>
   <Path name='/etc/ssh/ssh_host_key.pub'/>
   <Path name='/etc/ssh/sshd_config'/>
   <Path name='/etc/ssh/ssh_config'/>
   <Path name='/etc/ssh/ssh_known_hosts'/>
   <Group name='rpm'>
     <Package name='openssh'/>
     <Package name='openssh-askpass'/>
     <Service name='sshd'/>
     <Group name='fedora' >
        <Group name='fc4' negate='true'>
          <Package name='openssh-clients'/>
        </Group>
        <Package name='openssh-server'/>
     </Group>
   </Group>
   <Group name='deb'>
     <Package name='ssh'/>
     <Service name='ssh'/>
   </Group>
 </Bundle>
```

In this bundle, most of the entries are common to all systems. Clients in group **deb** get one extra package and service, while clients in group **rpm** get two extra packages and an extra service. In addition, clients in group **fedora** *and* group **rpm** get one extra package entries, unless they are not in the **fc4** group, in which case, they get an extra package. Notice that this file doesn't describe which versions of these entries that clients should get, only that they should get them. (Admittedly, this example is slightly contrived, but demonstrates how group entries can be used in bundles)

Group	Entry
all	/etc/ssh/ssh_host_dsa_key
all	/etc/ssh/ssh_host_rsa_key
all	/etc/ssh/ssh_host_dsa_key.pub
all	/etc/ssh/ssh_host_rsa_key.pub
all	/etc/ssh/ssh_host_key
all	/etc/ssh/ssh_host_key.pub
all	/etc/ssh/sshd_config
all	/etc/ssh/ssh_config
all	/etc/ssh/ssh_known_hosts
rpm	Package openssh
rpm	Package openssh-askpass
rpm	Service sshd
rpm and fedora	Package openssh-server
rpm and fedora and not fc4	Package openssh-clients
deb	Package ssh
deb	Service ssh

Genshi templates Genshi templates are used by adding a Genshi xml-style template to the Bundler directory with a `.genshi` file extension. Version 0.4 or newer of genshi is required.

Important: The `.genshi` file extension is required in order for the server to know that the Bundle should be rendered using Genshi.

Motivation Static Bundles have served us relatively well, but have a relatively weak set of interal per-client differentiation mechanisms. In static Bundles, the group hierarchy (from the perspective of the current client) is available for use via boolean constraints with negation. This notion does not mesh well with the use of Probes, which can result in freeform data. In the presence of probe results, clients can have a wide array of data, and rendering down to a boolean logic may not always be desirable. Moreover, while static Bundles allow entry inclusion or exclusion based on group memberships, they do not allow programatic entry rendering. Hence, Genshi templates not only provide more control options, but it also provide the ability to perform new entry rendering operations.

The Genshi templating system is used internally.

Use Bcfg uses the Genshi API for templates, and performs a XML format stream rendering of the template into an lxml entry, which is included in the client configuration. *Client metadata* is avilable inside of the template using the 'metadata' name. Note that only the markup Genshi template format can be used, as the target output format is XML.

A Genshi template looks much like a Bundler file, except the Bundle tag has an additional *xmlns:py* attribute. See the examples.

Altsrc

Fun and Profit using altsrc New in version 0.9.5. Altsrc is a generic, bcfg2-server-side mechanism for performing configuration entry name remapping for the purpose of data binding. Altsrc can be used as a parameter for any entry type, and can be used in any structure, including Bundler and Base.

Use Cases

- Equivalent configuration entries on different architectures with different names
- Mapping entries with the same name to different bind results in a configuration (two packages with the same name but different types)
- A single configuration entry across multiple specifications (multi-plugin, or multi-repo)

Examples

- Consider the case of /etc/hosts on linux and /etc/inet/hosts on solaris. These files contain the same data in the same format, and should typically be synchronized, however, exist in different locations. Classically, one would need to create one entry for each in Cfg or TCheetah and perform manual synchronization. Or, you could use symlinks and pray. Altsrc is driven from the bundle side. For example:

```
<Bundle name='netinfo'>
   <Group name='solaris'>
     <Path name='/etc/inet/hosts' altsrc='/etc/hosts'/>
   </Group>
   <Group name='linux'>
     <Path name='/etc/hosts'/>
   </Group>
</Bundle>
```

  In this case, when a solaris host gets the 'netinfo' bundle, it will get the first Path entry, which includes an altsrc parameter. This will cause the server to bind the entry as if it were a Path called `/etc/hosts`. This configuration entry is still called `/etc/inet/hosts`, and is installed as such.

- On encap systems, frequently multiple packages of the same name, but of different types will exist. For example, there might be an openssl encap package, and an openssl rpm package. This can be dealt with using a bundle like:

```
<Bundle name='openssl'>
   <Package name='openssl' altsrc='openssl-encap'/>
   <Package name='openssl' altsrc='openssl-rpm'/>
</Bundle>
```

  This bundle will bind data for the packages "openssl-encap" and "openssl-rpm", but will be delivered to the client with both packages named "openssl" with different types.

- Finally, consider the case where there exist complicated, but completely independent specifications for the same configuration entry but different groups of clients. The following bundle will allow the use of two different TCheetah templates /etc/firewall-rules-external and /etc/firewall-rules-internal for different clients based on their group membership.

```
<Bundle name='firewall'>
   ...
   <Group name='conduit'>
     <Path name='/etc/firewall-rules' altsrc='/etc/firewall-rules-external'/>
   </Group>
   <Group name='internal'>
     <Path name='/etc/firewall-rules' altsrc='/etc/firewall-rules-internal'/>
   </Group>
</Bundle>
```

- Consider the case where a variety of files can be constructed by a single template (TCheetah or TGenshi). It would be possible to copy this template into the proper location for each file, but that requires proper synchronization upon modification and knowing up front what the files will all be called. Instead, the following bundle allows the use of a single template for all proper config file instances.

```
<Bundle name='netconfig'>
  <Path name='/etc/sysconfig/network-scripts/ifcfg-eth0' altsrc='/etc/ifcfg-template
  <Path name='/etc/sysconfig/network-scripts/ifcfg-eth1' altsrc='/etc/ifcfg-template
  <Path name='/etc/sysconfig/network-scripts/ifcfg-eth2' altsrc='/etc/ifcfg-template
</Bundle>
```

Examples In some cases, configuration files need to include the client's hostname in their name. The following template produces such a config file entry.

```
<Bundle name='foo'  xmlns:py="http://genshi.edgewall.org/">
    <Path name='/etc/package-${metadata.hostname}'/>
</Bundle>
```

Depending on the circumstance, these configuration files can either be handled by individual entries in *Cfg*, *TCheetah*, or *TGenshi*, or can be mapped to a single entry by using the *Fun and Profit using altsrc* feature.

In this example, configuration file names are built using probed results from the client. getmac is a probe that gathers client MAC addresses and returns them in a newline delimited string.

```
<Bundle name='networkinterfaces' xmlns:py="http://genshi.edgewall.org/">
    <?python
      files = $metadata.Probes["getmacs"].split("\n")
    ?>
    <Path py:for="file in files" name="/etc/sysconfig/network/ifcfg-eth-${file}" altsrc='/e
</Bundle>
```

Note:

- The use of the altsrc directive causes all ifcfg files to be handled by the same plugin and entry.
- The <?python ?> blocks have only been available in genshi since 0.4 (http://genshi.edgewall.org/ticket/84)

If you want a file to be only on a per-client basis, you can use an if declaration:

```
<Bundle name='bacula' xmlns:py="http://genshi.edgewall.org/">
     <Path name="/etc/bacula/bconsole.conf"/>
     <Path name="/etc/bacula/bacula-fd.conf"/>
```

```
    <Path name="/etc/bacula/bacula-sd.conf"/>
    <Path py:if="metadata.hostname == 'foo.bar.com'" name="/etc/bacula/bacula-dir.conf"/>
</Bundle>
```

or alternately:

```
<Bundle name='bacula' xmlns:py="http://genshi.edgewall.org/">
    <Path name="/etc/bacula/bconsole.conf"/>
    <Path name="/etc/bacula/bacula-fd.conf"/>
    <Path name="/etc/bacula/bacula-sd.conf"/>
    <py:if test="metadata.hostname == 'foo.bar.com'">
        <Path name="/etc/bacula/bacula-dir.conf"/>
    </py:if>
</Bundle>
```

The latter form is preferred if the if block contains multiple files. While this example is simple, the test in the if block can in fact be any python statement.

Other examples Some simple examples of Bundles can be found in the example repository at the locations in the following table:

Bundle Name	Description
atxml	At bundle
bcfgxml	Bcfg2 client bundle
ntpxml	NTP bundle
sshxml	OpenSSH bundle
syslogxml	syslog bundle

In addition to the example repository, the following is a list of some more complex example Bundles.

kernel This is a rather complex Bundle for the Linux kernel from a system with a history of complexity. There are two kernel versions present on the systems at all times (the current and the previous), so the package names all contain versioning information. This includes kernel-specific modules for various specialties - `gm` for Myrinet boards, `gpfs` and `pvfs` for storage clients, and `nvidia` modules for machines with Nvidia cards. Note that only the `ia32` machines have Nvidia cards in them, and thus those entries only exist in that section.

It is easy to see that there is duplication of effort between the two architectures - both have the same `linux` package entry names, for example. This Bundle could be arranged in many different ways, some of which might be better than this one. Feel free to hack as needed.

```
<Bundle name='kernel' version='2.0'>
  <Group name='sles8'>
    <!-- ================== ia32 ==================== -->
    <Group name='ia32'>
      <Path name='/etc/lilo.conf'/>
      <Path name='/boot/vmlinuz'/>
      <Path name='/boot/initrd'/>
      <Path name='/boot/vmlinuz.old'/>
      <Path name='/boot/initrd.old'/>
      <PostInstall name='/sbin/lilo'/>
```

```
      <!-- Current kernel -->
      <Package name='linux-2.4.21-314.tg1'/>
      <Package name='linux-2.4.21-314.tg1-source'/>
      <!-- Old kernel -->
      <Package name='linux-2.4.21-309.tg1'/>
      <Group name='gm'>
        <Package name='gm-kernel-2.4.21-314.tg1'/>
        <Package name='gm-kernel-2.4.21-309.tg1'/>
      </Group>
      <Group name='storage-client'>
        <!-- Current kernel -->
        <Package name='gpfs-modules-2.4.21-314.tg1'/>
        <Package name='pvfs2-kernel-2.4.21-314.tg1'/>
        <!-- Old kernel -->
        <Package name='gpfs-modules-2.4.21-309.tg1'/>
        <Package name='pvfs2-kernel-2.4.21-309.tg1'/>
      </Group>
      <Group name='nvidia'>
        <Package name='NVIDIA-kernel-2.4.21-314.tg1'/>
        <Package name='NVIDIA-kernel-2.4.21-309.tg1'/>
      </Group>
    </Group>
    <!-- ================== ia64 =================== -->
    <Group name='ia64'>
      <Path name='/boot/efi/SuSE/elilo.conf'/>
      <!-- Current kernel -->
      <Package name='linux-2.4.21-314.tg1'/>
      <Package name='linux-2.4.21-314.tg1-source'/>
      <!-- Old kernel -->
      <Package name='linux-2.4.21-309.tg1'/>
      <Group name='gm'>
        <Package name='gm-kernel-2.4.21-314.tg1'/>
        <Package name='gm-kernel-2.4.21-309.tg1'/>
      </Group>
      <Group name='storage-client'>
        <!-- Current kernel -->
        <Package name='gpfs-modules-2.4.21-314.tg1'/>
        <Package name='pvfs2-kernel-2.4.21-314.tg1'/>
        <!-- Old kernel -->
        <Package name='gpfs-modules-2.4.21-309.tg1'/>
        <Package name='pvfs2-kernel-2.4.21-309.tg1'/>
      </Group>
    </Group>
  </Group>
</Bundle>
```

moab This is a fairly simple Bundle for the Moab workload manager.

```
<Bundle name='moab' version='2.0'>
  <Path name='/var/spool/moab'/>
  <Path name='/var/spool/moab/moab.cfg'/>
  <Group name='moab-server'>
```

```
    <Path name='/etc/init.d/moab'/>
    <Service name='moab'/>
  </Group>
</Bundle>
```

nagios A Bundle for the Nagios service. This Bundle installs all of our local Nagios plugins, takes into account that the SNMP package changed names between SLES 8 and SLES 9, and works on both the Nagios server and the clients.

```
<Bundle name='nagios-client' version='2.0'>
  <Group name='sles8'>
    <Package name='ucdsnmp'/>
  </Group>
  <Group name='sles9'>
    <Package name='net-snmp'/>
  </Group>
  <Package name='nagios-plugins'/>
  <Package name='perl-SNMP'/>
  <Package name='radiusclient' />
  <Package name='postgresql-libs' />
  <Package name='mysql-shared' />
  <Path name='/etc/hosts.deny'/>
  <Path name='/etc/services'/>
  <Path name='/etc/snmpd.conf'/>
  <Path name='/usr/lib/nagios/plugins/check_disks_scratchgpfs1.tg'/>
  <Path name='/usr/lib/nagios/plugins/check_fs.mds'/>
  <Path name='/usr/lib/nagios/plugins/check_gm_network.tg'/>
  <Path name='/usr/lib/nagios/plugins/check_gpfs_wan.tg'/>
  <Path name='/usr/lib/nagios/plugins/check_hung_jobs.tg'/>
  <Path name='/usr/lib/nagios/plugins/check_mem.mds'/>
  <Path name='/usr/lib/nagios/plugins/check_mem.tg'/>
  <Path name='/usr/lib/nagios/plugins/check_nvidia_acceleration.tg'/>
  <Path name='/usr/lib/nagios/plugins/check_os.mds'/>
  <Path name='/usr/lib/nagios/plugins/check_procinfo.mds'/>
  <Path name='/usr/lib/nagios/plugins/check_torque.tg'/>
  <Path name='/usr/lib/nagios/plugins/check_uname_r.tg'/>
  <Path name='/usr/lib/nagios/plugins/check_uname_r.tg.conf'/>
  <Service name='snmpd'/>
  <Group name='nagios-server'>
    <Package name='nagios'/>
    <Package name='nagios-devel'/>
    <Package name='nagios-www'/>
    <Path name='/etc/httpd/conf.d/nagios.conf'/>
    <Path name='/etc/nagios/cgi.cfg'/>
    <Path name='/etc/nagios/checkcommands.cfg'/>
    <Path name='/etc/nagios/nagios.cfg'/>
    <Path name='/etc/nagios/resource.cfg'/>
  </Group>
</Bundle>
```

Note: You may also want to have a look at the *NagiosGen* plugin.

ntp Despite its lack of groups, this Bundle controls both `ntp` servers and clients. It does this through the use of host-specific entries in the `Cfg` repository. It is left as an exercise for the reader to do this better through use of groups.

```
<Bundle name='ntp'>
  <Package name='xntp'/>
  <Path name='/etc/sysconfig/xntp'/>
  <Path name='/etc/sysconfig/clock'/>
  <Path name='/etc/ntp.conf'/>
  <Service name='xntpd'/>
</Bundle>
```

snmpd A simple bundle for a SNMP daemon with a package, a service and a configuration file.

```
<Bundle name="snmpd" version="3.0">
        <Package name="snmpd"/>
        <Service name="snmpd"/>
        <Path name="/etc/snmp/snmpd.conf"/>
</Bundle>
```

torque = torque.xml =

A longer Bundle that includes many group-specific entries.

```
<Bundle name='torque' version='1.0'>
  <Service name='nfs'/>
  <Service name='nfslock'/>
  <BoundPath type='directory' owner='root' group='root' perms='0755' name='/var/spool/torqu
  <BoundPath type='directory' owner='root' group='root' perms='0755' name='/var/spool/torqu
  <BoundPath type='directory' owner='root' group='root' perms='0755' name='/var/spool/torqu
  <Path name='/var/spool/torque/pbs_environment'/>
  <Path name='/var/spool/torque/torque.server'/>
  <Path name='/var/spool/torque/server_name'/>
  <Service name='jumbo'/>
  <Group name='torque-mom'>
    <Service name='torque_mom'/>
    <Path name='/etc/init.d/torque_mom'/>
    <BoundPath type='directory' owner='root' group='root' perms='0755' name='/var/spool/to
    <BoundPath type='directory' owner='root' group='root' perms='0755' name='/var/spool/to
    <BoundPath type='directory' owner='root' group='root' perms='0755' name='/var/spool/to
    <BoundPath type='directory' owner='root' group='root' perms='0755' name='/var/spool/to
    <BoundPath type='directory' owner='root' group='root' perms='0755' name='/var/spool/to
    <Path name='/var/spool/torque/mom_priv/config'/>
    <Path name='/var/spool/torque/mom_priv/prologue'/>
    <Path name='/var/spool/torque/mom_priv/epilogue'/>
  </Group>
  <Group name='torque-server'>
    <Service name='torque_server'/>
    <Path name='/etc/init.d/torque_server'/>
    <BoundPath type='directory' owner='root' group='root' perms='0755' name='/var/spool/to
    <BoundPath type='directory' owner='root' group='root' perms='0755' name='/var/spool/to
    <BoundPath type='directory' owner='root' group='root' perms='0755' name='/var/spool/to
```

```
    <BoundPath type='directory' owner='root' group='root' perms='0755' name='/var/spool/to
    <BoundPath type='directory' owner='root' group='root' perms='0755' name='/var/spool/to
    <BoundPath type='directory' owner='root' group='root' perms='0755' name='/var/spool/to
    <BoundPath type='directory' owner='root' group='root' perms='0755' name='/var/spool/to
    <BoundPath type='directory' owner='root' group='root' perms='0755' name='/var/spool/to
    <BoundPath type='directory' owner='root' group='root' perms='0755' name='/var/spool/to
  </Group>
</Bundle>
```

yp = yp.xml =

Note that this Bundle includes **Group** sections. Toplevel elements go to anybody that includes this Bundle, but clients that belong to the **yp-client** and **yp-server** groups get their own specialized treatment too.

```
<Bundle name='yp' version='2.0'>
  <Package name='yp-tools'/>
  <Path name='/etc/nsswitch.conf'/>
  <Path name='/etc/yp.conf'/>
  <Path name='/etc/defaultdomain'/>
  <Group name='yp-client'>
    <Package name='ypbind'/>
    <Service name='ypbind'/>
    <Service name='portmap'/>
  </Group>
  <Group name='yp-server'>
    <Package name='ypserv'/>
    <Service name='ypserv'/>
    <Path name='/etc/ypserv.conf'/>
  </Group>
</Bundle>
```

Fun and Profit using altsrc

New in version 0.9.5. Altsrc is a generic, bcfg2-server-side mechanism for performing configuration entry name remapping for the purpose of data binding. Altsrc can be used as a parameter for any entry type, and can be used in any structure, including Bundler and Base.

Use Cases

- Equivalent configuration entries on different architectures with different names
- Mapping entries with the same name to different bind results in a configuration (two packages with the same name but different types)
- A single configuration entry across multiple specifications (multi-plugin, or multi-repo)

Examples

- Consider the case of /etc/hosts on linux and /etc/inet/hosts on solaris. These files contain the same data in the same format, and should typically be synchronized, however, exist in different locations.

Classically, one would need to create one entry for each in Cfg or TCheetah and perform manual synchronization. Or, you could use symlinks and pray. Altsrc is driven from the bundle side. For example:

```
<Bundle name='netinfo'>
   <Group name='solaris'>
     <Path name='/etc/inet/hosts' altsrc='/etc/hosts'/>
   </Group>
   <Group name='linux'>
     <Path name='/etc/hosts'/>
   </Group>
</Bundle>
```

In this case, when a solaris host gets the 'netinfo' bundle, it will get the first Path entry, which includes an altsrc parameter. This will cause the server to bind the entry as if it were a Path called `/etc/hosts`. This configuration entry is still called `/etc/inet/hosts`, and is installed as such.

- On encap systems, frequently multiple packages of the same name, but of different types will exist. For example, there might be an openssl encap package, and an openssl rpm package. This can be dealt with using a bundle like:

```
<Bundle name='openssl'>
   <Package name='openssl' altsrc='openssl-encap'/>
   <Package name='openssl' altsrc='openssl-rpm'/>
</Bundle>
```

This bundle will bind data for the packages "openssl-encap" and "openssl-rpm", but will be delivered to the client with both packages named "openssl" with different types.

- Finally, consider the case where there exist complicated, but completely independent specifications for the same configuration entry but different groups of clients. The following bundle will allow the use of two different TCheetah templates /etc/firewall-rules-external and /etc/firewall-rules-internal for different clients based on their group membership.

```
<Bundle name='firewall'>
   ...
   <Group name='conduit'>
     <Path name='/etc/firewall-rules' altsrc='/etc/firewall-rules-external'/>
   </Group>
   <Group name='internal'>
     <Path name='/etc/firewall-rules' altsrc='/etc/firewall-rules-internal'/>
   </Group>
</Bundle>
```

- Consider the case where a variety of files can be constructed by a single template (TCheetah or TGenshi). It would be possible to copy this template into the proper location for each file, but that requires proper synchronization upon modification and knowing up front what the files will all be called. Instead, the following bundle allows the use of a single template for all proper config file instances.

```
<Bundle name='netconfig'>
  <Path name='/etc/sysconfig/network-scripts/ifcfg-eth0' altsrc='/etc/ifcfg-template
  <Path name='/etc/sysconfig/network-scripts/ifcfg-eth1' altsrc='/etc/ifcfg-template
  <Path name='/etc/sysconfig/network-scripts/ifcfg-eth2' altsrc='/etc/ifcfg-template
</Bundle>
```

Base

The Base plugin is a structure plugin that provides the ability to add lists of unrelated entries into client configuration entry inventories.

Base works much like Bundler in its file format. The main difference between Base and Bundler is that Base files are included in all clients' configuration whereas bundles must be included explicitly in your Metadata. See the *Bundler* page for details.

If you have lots of unconnected items (for instance: software packages whose configuration wasn't modified, and that are also not depended on by other packages; or single directories or files not belonging to a package), using Bundles in Metadata would clutter or enlarge your `Metadata/groups.xml` file, because they all would need to be explicitly specified. `Base/` on the other hand is the perfect place to put these items.

Without using Base, you would be forced to put them directly into your group definitions in `groups.xml`, either as many small bundles (substantially enlarging it) or into something like `Bundler/unrelated-entries.xml`. Using the latter is especially bad if you mix packages and services in your Bundle, since for any updated package in that bundle, the now-related services would be restarted.

The Base entries can still be assigned based on group membership, but when they aren't part of a group, each and every client gets the entry. So Base is also a great place to put entries that a large number of your clients will get.

For example, you could have a file `Base/packages.xml`

```
<Base>
    <Package name='acpid'/>
    <Package name='auditd'/>
    [...]
    <Group name='openSUSE11.2'>
        <Package name='syslog-ng'/>
    </Group>
    <Group name='openSUSE11.3'>
        <Package name='rsyslog'/>
    </Group>
    [...]
    <Package name='zlib'/>
</Base>
```

Note: You don't have to reference to the files in Base from anywhere. As long as you include `Base` in your `plugins = ...` line in `bcfg2.conf`, these are included automatically.

Note: Your Base files have to match the pattern `Base/*.xml` to be included.

The decision when to use Base and when to use Bundler depends on the configuration entry in question, and what you are trying to achieve.

Base is mainly used for cases where you don't want/need to explicitly include particular configuration items. Let's say all your machines are various linux distributions. In this case, you may want to manage the `/etc/hosts` file using Base instead of Bundler since you will not have to include any Bundles in your Metadata. However, you could alternatively have a base 'linux' group that all the clients inherit which includes a 'linux' Bundle with the `/etc/hosts` configuration entry.

Each of these plugins has a corresponding subdirectory with the same name in the Bcfg2 repository.

Literal Configuration (Generators)

TGenshi

This page documents the TGenshi plugin. This plugin works with version 0.4 and newer of the genshi library.

The TGenshi plugin allows you to use the Genshi templating system to create files, instead of the various diff-based methods offered by the Cfg plugin. It also allows you to include the results of probes executed on the client in the created files.

To begin, you will need to download and install the Genshi templating engine.

To install on CentOS or RHEL 5, run:

```
sudo yum install python-genshi
```

Once it is installed, you can enable it by adding `TGenshi` to the generators line in `/etc/bcfg2.conf` on your Bcfg server. For example:

```
generators = SSHbase,Cfg,Pkgmgr,Svcmgr,Rules,TGenshi
```

The TGenshi plugin makes use of a Cfg-like directory structure located in in a TGenshi subdirectory of your repository, usually `/var/lib/bcfg2/TGenshi`. Each file has a directory containing two file types, template and info. Templates are named according to the genshi format used; template.txt uses the genshi text format, and template.xml uses the XML format.

If used with Genshi 0.5 or later the plugin also supports the new style text template format for files named template.newtxt. One of the advantages of the new format is that it does not use # as a command delimiter, making it easier to utilize for configuration files that use # as a comment character.

Only one template format may be used per file served. Info files are identical to those used in `Cfg`, and `info.xml` files are supported.

Inside of templates

- metadata is the client's metadata
- properties.properties is an xml document of unstructured data

See the genshi documentation for examples of Genshi syntax.

Examples: Old Genshi Syntax Genshi's web pages recommend against using this syntax, as it may disappear from future releases.

Group Negation Templates are also useful for cases where more sophisticated boolean operations than those supported by Cfg are needed. For example, the template:

```
#if "ypbound" in metadata.groups and "workstation" in metadata.groups
client is ypbound workstation
#end
#if "ubuntu" not in metadata.groups and "desktop" in metadata.groups
client is a desktop, but not an ubuntu desktop
#end
```

Produces:

```
<Path type="file" name="/bar.conf" owner="root" perms="0644" group="root">client is ypbound
client is a desktop, but not an ubuntu desktop
</Path>
```

This flexibility provides the ability to build much more compact and succinct definitions of configuration contents than Cfg can.

File permissions File permissions for entries handled by TGenshi are controlled via the use of *Info* files. Note that you **cannot** use both a Permissions entry and a Path entry to handle the same file.

Error handling Situations may arrise where a templated file cannot be generated due to missing or incomplete information. A TemplateError can be raised to force a bind failure and prevent sending an incomplete file to the client. For example, this template:

```
{% python
    from genshi.template import TemplateError
    grp = None
    for g in metadata.groups:
        if g.startswith('ganglia-gmond-'):
            grp = g
            break
    else:
        raise TemplateError, "Missing group"
%}\
```

will fail to bind if the client is not a member of a group starting with "ganglia-gmond-". The syslogs on the server will contain this message:

```
bcfg2-server[5957]: Genshi template error: Missing group
bcfg2-server[5957]: Failed to bind entry: Path /etc/ganglia/gmond.conf
```

indicating the bind failure and message raised with the TemplateError.

FAQs Question

How do I escape the $ (dollar sign) in a TGenshi text template? For example, if I want to include SVN (subversion) keywords like Id or $HeadURL$ in TGenshi-generated files, or am templating a bourne shell (sh/bash) script or Makefile (make).

Answer

Use $$ (double dollar sign) to output a literal $ (dollarsign) in a TGenshi text template. So instead of Id, you'd use $$Id$$. See also Genshi tickets #282: Document $$ escape convention and #283: Allow for redefinition of template syntax per-file.

Examples

bcfg2-cron As submitted by Kamil Kisiel

The following is my `/etc/cron.d/bcfg2` file. It uses the python random module seeded with the client hostname to generate a random time for the client to check in. The hostname seed ensures the generated file is the same each time the client checks in. This cron file helps to distribute the load on the Bcfg2 server since not all machines are checking in at the same time.:

```
{% python
from genshi.builder import tag
import random
random.seed(metadata.hostname)
%}\
${random.randint(0,60)} * * * *    root    /usr/sbin/bcfg2 &> /dev/null
```

You can apply the same concept to the other time fields by adding another `${random.randint()}` call.

clientsxml As submitted by dclark

Here is an example of maintaining the bcfg2 server's `/var/lib/bcfg2/Metadata/clients.xml` file using TGenshi.

There are two main advantages:

1. Password storage is centralized in the `Properties/passwords.xml` file this helps maintain consistency, makes changing passwords easier, and also makes it easier to share your configurations with other sites/people.
2. You can template the file using Genshi's *{% def %}* syntax, which makes *clients.xml* much more readable. An important thing to note is how the *name* variable is handled - when just referring to it the standard *${name}* syntax is used, but when it is used as a variable in the expression to get the password, *password="${metadata.Properties['passwords.xml'].find('password').find('bcfg2-client').find(name).text}"*, it is just referred to as *name*.

There is the disadvantage that sometimes 2 passes will be needed to get to a consistent state.

Possible improvements:

1. Wrapper for bcfg2 client runs on the bcfg2 server, perhaps using a call to *bcfg2-info buildfile*, so clients.xml is always generated before everything else happens (since the state of clients.xml can influence everything else bcfg2-server does).
2. We really don't care what the client passwords are, just that they exist, so instead of listing them a master password combined with some kind of one-way hash based on the *name* might make more sense, and make `Properties/passwords.xml` easier to maintain.

- TGenshi/var/lib/bcfg2/Metadata/clients.xml/template.newtxt:

```
<!-- TGenshi/var/lib/bcfg2/Metadata/clients.xml/template.newtxt -->
<!-- Do not edit this file directly - edit only the above template -->

{# Doc: http://bcfg2.org/wiki/Authentication #}\
{% def static(profile,name,address) %}
    <Client
        profile="${profile}"
        name="${name}"
        uuid="${name}"
        password="${metadata.Properties['passwords.xml'].find('password').find('bcfg
        address="${address}"
        location="fixed"
        secure="true"
    />\
{% end %}\
{% def dynamic(profile,name) %}
    <Client
        profile="${profile}"
        name="${name}"
        uuid="${name}"
        password="${metadata.Properties['passwords.xml'].find('password').find('bcfg
        location="floating"
        secure="true"
    />\
{% end %}\
<Clients version="3.0">\
    ${static('group-server-collab','campaigns.example.com','192.168.111.1')}
    ${static('group-server-collab','info.office.example.com','192.168.111.2')}
    ${static('group-server-config','config.example.com','192.168.111.3')}
    ${dynamic('group-project-membercard','membercard')}
    ${dynamic('group-person-somename','somename.office.example.com')}
</Clients>
```

- Properties/passwords.xml snippit:

```
<Properties>
        <password>
                <bcfg2-client>
                        <campaigns.example.com>FAKEpassword1</campaigns.example.com>
                        <info.office.example.com>FAKEpassword2</info.office.example.
                        <config.example.com>FAKEpassword3</config.example.com>
                        <membercard>FAKEpassword4</membercard>
                        <somename.office.example.com>FAKEpassword5</somename.office.
                </bcfg2-client>
        </password>
</Properties>
```

ganglia Another interesting example of **TGenshi** templating is to automatically generate `gmond`/`gmetad` configuration files. The idea is that each cluster is headless: it communicates with the rest of the cluster members on an isolated multicast IP address and port. Any of the cluster members is therefore isolated on that particular ip/port pair. Additionally, each `gmond` instance **also** listens on UDP. This allows for any of the cluster members to be polled for information on the entire cluster!

The second part of the trick is in `gmetad.conf`. Here, we dynamically generate a list of clusters (based on profiles names) and a list of members to poll (based on the clients in said profiles). As the number of profiles and client grows, this list will grow automatically as well. When a new host is added, `gmetad` will receive an updated configuration and act accordingly.

There **is** one caveat though. The `gmetad.conf` parser is hard coded to read 16 arguments per `data_source` line. If you have more than 15 nodes in a cluster, you will see a warning in the logs. You can either ignore it, or truncate the list to the first 15 members.

In our environment, a profile is a one to one match with the role of that particular host. You can also do this based on groups, or any other client property.

Bundler/ganglia.xml

```
<Bundle name='ganglia' version='2.0' revision='$Revision$' origin='$HeadURL$' >
    <Package name='ganglia-gmond' />
    <Package name='ganglia-gmond-modules-python' />
    <Path name='/etc/ganglia/gmond.conf' />
    <Service name='gmond' />
    <Action name='gmond-reload' />

    <Group name='gmetad-server'>
      <Package name='ganglia-gmetad'/>
      <Package name='ganglia-web'/>
      <Package name='rrdtool'/>
      <Path name='/etc/ganglia/gmetad.conf' />
      <Service name='gmetad' />
    </Group>
</Bundle>
```

Rules/services-ganglia.xml

```
<Rules priority='10' revision='$Revision$' origin='$HeadURL$' >
    <Service name='gmond' type='chkconfig' status='on' />
    <Group name='gmetad-server'>
        <Service name='gmetad' type='chkconfig' status='on' />
    </Group>
</Rules>
```

TGenshi/etc/ganglia/gmetad.conf/template.newtxt

```
{% python
   client_metadata =  metadata.query.all()
   profile_array = {}
   seen = []
   for item in client_metadata:
       if item.profile not in seen:
           seen.append(item.profile)
           profile_array[item.profile]=[]
       profile_array[item.profile].append(item.hostname)
   seen.sort()
%}\
```

```
gridname "Our Grid"

{% for profile in seen %}
data_source "${profile}" \
{% for host in profile_array[profile] %}\
${host} \
{% end %}\
{% end %}

rrd_rootdir "/var/lib/ganglia/rrds"
```

TGenshi/etc/ganglia/gmond.conf/template.newtxt

```
{% python
from genshi.builder import tag
import random
random.seed(metadata.profile)
last_octet=random.randint(2,254)
%}\
/*
 $$Id$$
 $$HeadURL$$
*/

/* This configuration is as close to 2.5.x default behavior as possible
   The values closely match ./gmond/metric.h definitions in 2.5.x */
globals {
  daemonize = yes
  setuid = yes
  user = nobody
  debug_level = 0
  max_udp_msg_len = 1472
  mute = no
  deaf = no
  host_dmax = 1800 /* 30 minutes */
  cleanup_threshold = 604800 /*secs=1 week */
  gexec = no
  send_metadata_interval = 0
}

/* If a cluster attribute is specified, then all gmond hosts are wrapped inside
 * of a <CLUSTER> tag.  If you do not specify a cluster tag, then all <HOSTS> will
 * NOT be wrapped inside of a <CLUSTER> tag. */
cluster {
  name = "${metadata.profile}"
  owner = "user@company.net"
  latlong = "unspecified"
  url = "unspecified"
}

/* The host section describes attributes of the host, like the location */
host {
```

```
  location = "unspecified"
}

/* Feel free to specify as many udp_send_channels as you like.  Gmond
   used to only support having a single channel */
udp_send_channel {
  host = ${metadata.hostname}
  port = 8649
}
udp_send_channel {
  mcast_join = 239.2.11.${last_octet}
  port = 8649
  ttl = 1
}

/* You can specify as many udp_recv_channels as you like as well. */
udp_recv_channel {
  port = 8649
  bind = ${metadata.hostname}
}
udp_recv_channel {
  mcast_join = 239.2.11.${last_octet}
  bind       = 239.2.11.${last_octet}
  port = 8649
}

/* You can specify as many tcp_accept_channels as you like to share
   an xml description of the state of the cluster */
tcp_accept_channel {
  port = 8649
}

/* Each metrics module that is referenced by gmond must be specified and
   loaded. If the module has been statically linked with gmond, it does not
   require a load path. However all dynamically loadable modules must include
   a load path. */
modules {
/* [snip] */
```

grubconf Automate the build of grub.conf based on probe data. In this case, we take the results from three probes, serial-console-speed, grub-serial-order, and current-kernel to fill in a few variables. In addition, we want at least two entries set up for the kernel: a multiuser and a single user option.

```
# grub.conf generated by anaconda
#
# Note that you do not have to rerun grub after making changes to this file
# NOTICE:  You have a /boot partition.  This means that
#          all kernel and initrd paths are relative to /boot/, eg.
#          root (hd0,0)
#          kernel /vmlinuz-version ro root=/dev/VolGroup00/LogVol00
#          initrd /initrd-version.img
#boot=/dev/sda
```

```
default=0
timeout=5
serial --unit=0 --speed=${metadata.Probes['serial-console-speed']}
terminal --timeout=5 ${metadata.Probes['grub-serial-order']}

{% for kernbootoption in ["", "single"] %}\
title Red Hat Enterprise Linux Server (${metadata.Probes['current-kernel']})) ${kernbootopt
        root (hd0,0)
        kernel /vmlinuz-${metadata.Probes['current-kernel']} ro root=/dev/VolGroup00/LogVol
        initrd /initrd-${metadata.Probes['current-kernel']}.img
{% end %}\
```

hosts This is an example of creating `/etc/hosts` based on metadata.hostname:

```
# Do not remove the following line, or various programs
# that require network functionality will fail.
127.0.0.1       localhost.localdomain localhost
::1             localhost6.localdomain6 localhost6
{% python
import socket
import re
ip = socket.gethostbyname(metadata.hostname)

shortname = re.split("\.", metadata.hostname)
%}\
${ip} ${metadata.hostname} ${shortname[0]}
```

iptables

- Setup a TGenshi base iptables file that contains the basic rules you want every host to have
- Create a custom dir that has group and host specific rules you want to apply
- To be safe you should have a client side IptablesDeadmanScript if you intend on having bcfg2 bounce iptables upon rule updates

Note: When updating files in the `includes` directory, you will need to *touch* the TGenshi template to regenerate the template contents.

/repository/TGenshi/etc/sysconfig/iptables/template.newtxt

```
{% python
    from genshi.builder import tag
    import os,sys
    import Bcfg2.Options

    opts = { 'repo': Bcfg2.Options.SERVER_REPOSITORY }
    setup = Bcfg2.Options.OptionParser(opts)
    setup.parse('--')
    repo = setup['repo']
    basedir = '%s' % (repo)
```

```
    # for instance: /var/lib/bcfg2/custom/etc/sysconfig/iptables/
    bcfg2BaseDir = basedir + '/includes' + name + '/'


    def checkHostFile(hostName, type):
        fileName = bcfg2BaseDir + type + '.H_' + hostName
        if os.path.isfile(fileName)==True :
           return fileName
        else:
           return fileName

    def checkGroupFile(groupName, type):
        fileName = bcfg2BaseDir + type + '.G_' + groupName
        if os.path.isfile(fileName)==True :
           return fileName
        else:
           return fileName

%}\
# BCFG2 GENERATED IPTABLES
# DO NOT CHANGE THIS
# $$Id$$
# $$HeadURL$$
# Templates live in ${bcfg2BaseDir}
# Manual customization of this file will get reverted.
# ----------------------------- FILTER -------------------------------- #
# Default CHAINS for FILTER:
*filter
:INPUT DROP [0:0]
:FORWARD DROP [0:0]
:OUTPUT ACCEPT [0:0]
:NO-SMTP - [0:0]

#Default rules
#discard malicious packets
-A INPUT -p tcp --tcp-flags ALL ACK,RST,SYN,FIN -j DROP
-A INPUT -p tcp --tcp-flags SYN,FIN SYN,FIN -j DROP
-A INPUT -p tcp --tcp-flags SYN,RST SYN,RST -j DROP
#Allow incoming ICMP
-A INPUT -p icmp -m icmp -j ACCEPT
#Accept localhost traffic
-A INPUT -i lo -j ACCEPT
# Allow already established sessions to remain
-A INPUT -m state --state RELATED,ESTABLISHED -j ACCEPT

# Deny inbound SMTP delivery (still allows outbound connections)
-A INPUT -m state --state NEW -m tcp -p tcp --tcp-flags FIN,SYN,RST,ACK SYN --dport 25 -j N
-A NO-SMTP -j LOG --log-prefix " Incoming SMTP (denied) "
-A NO-SMTP -j DROP

# Allow SSH Access
-A INPUT  -p tcp -m state --state NEW -m tcp --tcp-flags FIN,SYN,RST,ACK SYN --dport 22 -j
-A SSH -s 192.0.0.0/255.0.0.0  -j ACCEPT
```

```
# Allow Ganglia Access
-A INPUT -m state --state NEW -m tcp -p tcp --tcp-flags FIN,SYN,RST,ACK SYN --src 192.168.1
# Gmetad access to gmond
-A INPUT -m state --state NEW -m tcp -p tcp --tcp-flags FIN,SYN,RST,ACK SYN --src 192.168.1
# Gmond UDP multicast
-A INPUT -m state --state NEW -m udp -p udp --dport 8649 -j ACCEPT

{% if metadata.groups %}\
# group custom FILTER rules:
{% for group in metadata.groups %}\
{% include ${checkGroupFile(group,'custom-filter')} %}\
{% end %}\
{% end %}\

# host-specific FILTER rules:
{% include ${checkHostFile(metadata.hostname, 'custom-filter')} %}\

COMMIT
# ------------------------------- NAT ----------------------------------- #
*nat

# Default CHAINS for NAT:
:PREROUTING ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
:POSTROUTING ACCEPT [0:0]

{% if metadata.groups %}\
# group NAT for PREROUTING:
{% for group in metadata.groups %}\
{% include ${checkGroupFile(group,'nat-prerouting')} %}\
{% end %}\
{% end %}\

{% if metadata.groups %}\
# group NAT for OUTPUT:
{% for group in metadata.groups %}\
{% include ${checkGroupFile(group,'nat-output')} %}\
{% end %}\
{% end %}\

{% if metadata.groups %}\
# group NAT for POSTROUTING:
{% for group in metadata.groups %}\
{% include ${checkGroupFile(group,'nat-postrouting')} %}\
{% end %}\
{% end %}\

{% if metadata.groups %}\
# group custom NAT rules:
{% for group in metadata.groups %}\
{% include ${checkGroupFile(group,'custom-nat')} %}\
{% end %}\
{% end %}\
```

```
# host-specific NAT ruls:
{% include ${checkHostFile(metadata.hostname, 'custom-nat')} %}\
COMMIT
# ----------------------------- MANGLE -------------------------------- #
*mangle

# Default CHAINS for MANGLE:
:PREROUTING ACCEPT [0:0]
:INPUT ACCEPT [0:0]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
:POSTROUTING ACCEPT [0:0]

{% if metadata.groups %}\
# group MANGLE for PREROUTING:
{% for group in metadata.groups %}\
{% include ${checkGroupFile(group,'mangle-prerouting')} %}\
{% end %}\
{% end %}\

{% if metadata.groups %}\
# group MANGLE for INPUT:
{% for group in metadata.groups %}\
{% include ${checkGroupFile(group,'mangle-input')} %}\
{% end %}\
{% end %}\

{% if metadata.groups %}\
# group MANGLE for FORWARD:
{% for group in metadata.groups %}\
{% include ${checkGroupFile(group,'mangle-forward')} %}\
{% end %}\
{% end %}\

{% if metadata.groups %}\
# group MANGLE for OUTPUT:
{% for group in metadata.groups %}\
{% include ${checkGroupFile(group,'mangle-output')} %}\
{% end %}\
{% end %}\

{% if metadata.groups %}\
# group MANGLE for POSTROUTING rules:
{% for group in metadata.groups %}\
{% include ${checkGroupFile(group,'mangle-postrouting')} %}\
{% end %}\
{% end %}\

{% if metadata.groups %}\
# group custom MANGLE rules:
{% for group in metadata.groups %}\
{% include ${checkGroupFile(group,'custom-mangle')} %}\
{% end %}\
```

```
{% end %}\

# host-specific MANGLE rules:
{% include ${checkHostFile(metadata.hostname, 'custom-mangle')} %}\
COMMIT
```

/var/lib/bcfg2/custom/etc/sysconfig/iptables/custom-filter.G_mysql-server

```
:MYSQL - [0:0]
-A INPUT  -p tcp -m state --state NEW -m tcp --dport 3306 --tcp-flags FIN,SYN,RST,ACK SYN -
-A MYSQL -s 192.168.0.0/255.0.0.0  -j ACCEPT
```

For a host that is in the mysql-server group you get an iptables file that looks like the following:

```
# BCFG2 GENERATED IPTABLES
# DO NOT CHANGE THIS
# $Id: template.newtxt 5402 2009-08-19 22:50:06Z unixmouse$
# $HeadURL: https://svn.fakecompany.net/bcfg2/trunk/repository/TGenshi/etc/sysconfig/iptabl
# Templates live in /var/lib/bcfg2/custom/etc/sysconfig/iptables/
# Manual customization of this file will get reverted.
# ----------------------------- FILTER --------------------------------- #
# Default CHAINS for FILTER:
*filter
:INPUT DROP [0:0]
:FORWARD DROP [0:0]
:OUTPUT ACCEPT [0:0]
:NO-SMTP - [0:0]

#Default rules
#discard malicious packets
-A INPUT -p tcp --tcp-flags ALL ACK,RST,SYN,FIN -j DROP
-A INPUT -p tcp --tcp-flags SYN,FIN SYN,FIN -j DROP
-A INPUT -p tcp --tcp-flags SYN,RST SYN,RST -j DROP
# Allow incoming ICMP
-A INPUT -p icmp -m icmp -j ACCEPT
# Accept localhost traffic
-A INPUT -i lo -j ACCEPT
# Allow already established sessions to remain
-A INPUT -m state --state RELATED,ESTABLISHED -j ACCEPT

# Deny inbound SMTP delivery (still allows outbound connections)
-A INPUT -m state --state NEW -m tcp -p tcp --tcp-flags FIN,SYN,RST,ACK SYN --dport 25 -j N
-A NO-SMTP -j LOG --log-prefix " Incoming SMTP (denied) "
-A NO-SMTP -j DROP

# Allow SSH Access
:SSH - [0:0]
-A INPUT  -p tcp -m state --state NEW -m tcp --tcp-flags FIN,SYN,RST,ACK SYN --dport 22 -j
-A SSH -s 192.168.0.0/255.0.0.0  -j ACCEPT

# Allow Ganglia Access
-A INPUT -m state --state NEW -m tcp -p tcp --tcp-flags FIN,SYN,RST,ACK SYN --src 192.168.1
#Gmetad access to gmond
```

```
-A INPUT -m state --state NEW -m tcp -p tcp --tcp-flags FIN,SYN,RST,ACK SYN --src 192.168.1
#Gmond UDP multicast
-A INPUT -m state --state NEW -m udp -p udp --dport 8649 -j ACCEPT

# group custom FILTER rules:
:MYSQL - [0:0]
-A INPUT  -p tcp -m state --state NEW -m tcp --dport 3306 --tcp-flags FIN,SYN,RST,ACK SYN -
-A MYSQL -s 192.168.0.0/255.0.0.0  -j ACCEPT

# host-specific FILTER rules:

COMMIT
# ------------------------------ NAT ---------------------------------- #
*nat

# Default CHAINS for NAT:
:PREROUTING ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
:POSTROUTING ACCEPT [0:0]

# group NAT for PREROUTING:

# group NAT for OUTPUT:

# group NAT for POSTROUTING:

# group custom NAT rules:

# host-specific NAT rules:
COMMIT
# ----------------------------- MANGLE ------------------------------- #
*mangle

# Default CHAINS for MANGLE:
:PREROUTING ACCEPT [0:0]
:INPUT ACCEPT [0:0]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
:POSTROUTING ACCEPT [0:0]

# group MANGLE for PREROUTING:

# group MANGLE for INPUT:
# group MANGLE for FORWARD:

# group MANGLE for OUTPUT:

# group MANGLE for POSTROUTING rules:

# group custom MANGLE rules:

# host-specific MANGLE rules:
COMMIT
```

motd The following template automatically generates a MOTD (message of the day) file that describes the system in terms of its Bcfg2 metadata and probe responses. It conditionally displays groups, categories, and probe responses, if there exists any data for them.

New Style of TGenshi This is the preferred way of creating TGenshi contents. It requires Genshi 0.5 or later.

On the Bcfg2 server Where, **$bcfg2** is your Bcfg2 repository on your Bcfg2 server, the following files need to be created:

```
$bcfg2/TGenshi/etc/motd/info.xml
$bcfg2/TGenshi/etc/motd/template.newtxt
```

The contents of `motd/template.newtxt` could be something like this:

```
------------------------------------------------------------------------
                    GOALS FOR SERVER MANGED BY BCFG2
------------------------------------------------------------------------
Hostname is ${metadata.hostname}

Groups:
{% for group in metadata.groups %}\
 * ${group}
{% end %}\

{% if metadata.categories %}\
Categories:
{% for category in metadata.categories %}\
 * ${category}
{% end %}\
{% end %}\


{% if metadata.Probes %}\
Probes:
{% for probe, value in metadata.Probes.iteritems() %}\
 * ${probe} \
   ${value}
{% end %}\
{% end %}\

------------------------------------------------------------------------
                         ITOPS MOTD
------------------------------------------------------------------------
Please create a Ticket for any system level changes you need from IT.
```

This template gets the hostname, groups membership of the host, categories of the host (if any), and result of probes on the host (if any). The template formats this in with a header and footer that makes it visually more appealing.

A `motd/info.xml` file isn't strictly needed, because `/etc/motd` has the Bcfg2 default permissions (i.e. root:root 0644), but it can be included for completeness.

Output One possible output of this template would be the following:

```
------------------------------------------------------------------------
                    GOALS FOR SERVER MANGED BY BCFG2
------------------------------------------------------------------------
Hostname is cobra.example.com

Groups:
 * oracle-server
 * centos5-5.2
 * centos5
 * redhat
 * x86_64
 * sys-vmware

Categories:
 * os-variant
 * os
 * database-server
 * os-version


Probes:
 * arch    x86_64
 * network    intranet_network
 * diskspace    Filesystem            Size  Used Avail Use% Mounted on
/dev/mapper/VolGroup00-LogVol00
                       18G  2.1G   15G  13% /
/dev/sda1              99M   13M   82M  13% /boot
tmpfs                 3.8G     0  3.8G   0% /dev/shm
/dev/mapper/mhcdbo-clear
                      1.5T  198M  1.5T   1% /mnt/san-oracle
 * virtual    vmware

------------------------------------------------------------------------
                        IT MOTD
------------------------------------------------------------------------
Please create a Ticket for any system level changes you need from IT.
```

Taking it to the next level One way to make this even more useful, is to only include the result of certain probes. It would also be a nice feature to be able to include customer messages on a host or group level.

Old Style of TGenshi The following is a way to do the same thing using the older, it-may-be-depreciated, style of Genshi (pre-0.5).:

```
Hostname is $metadata.hostname

Groups:
#for group in metadata.groups
 * $group
#end
```

```
#if metadata.categories
Categories:
#for category in metadata.categories
 * $category
#end
#end

#if metadata.probes
Probes:
#for probe, value in metadata.probes.iteritems()
 * $probe $value
#end
#end
```

This template results in:

```
> buildfile /bar.conf ubik3
<Path name="/bar.conf" type="file" owner="root" perms="0644" group="root">Hostname is ubik

Groups:
 * desktop
 * computeserver
 * mcs-base
 * ypbound
 * workstation
 * mysql-4
 * debian-sarge-base
 * debian-sarge
 * base
 * debian

Categories:
 * noyp
 * mysql

</Path>
```

mycnf The following template generates a `server-id` based on the last two numeric parts of the IP address. The "slave" portion of the configuration only applies to machines in the "slave" group.:

```
{% python
from genshi.builder import tag
import socket
parts = socket.gethostbyname(metadata.hostname).split('.')
server_id = parts[2] + parts[3]
%}\
[mysqld]

# [snip]

server-id = ${server_id}
```

```
# Replication configuration

{% if "slave" in metadata.groups %}\
relay-log = /data01/mysql/log/mysql-relay-bin
log-slave-updates = 1
{% end %}\
sync-binlog = 1
#read-only = 1
#report-host = <server fqdn>

# [snip]
```

test FIXME: This example needs to be retested with new Properties plugin.

As submitted by dclark

This file just shows you what's available. It assumes a `/var/lib/bcfg2/Properties/test.xml` file with an entry like this:

```
#!text/xml
<Properties>
  <password>
    <bcfg2>fakeBCFG2password</bcfg2>
  </password>
</Properties>

Hostname is ${metadata.hostname}

Groups:
{% for group in metadata.groups %}\
${group} \
{% end %}\

{% if metadata.categories %}\
Categories:
{% for category in metadata.categories %}\
${category} \
{% end %}\
{% end %}\

{% if metadata.Probes %}\
Probes:
{% for probe, value in metadata.Probes.iteritems() %}\
$probe $value
{% end %}\
{% end %}\

Two main ways to get the same property value:
${metadata.Properties['test.xml'].find('password').find('bcfg2').text}
${metadata.Properties['test.xml'].xpath('password/bcfg2')[0].text}

One way to get information about metadata and properties:
```

```
dir(metadata):
{% for var in dir(metadata) %}\
${var} \
{% end %}

dir(properties):
{% for var in dir(properties) %}\
${var} \
{% end %}

dir(properties.entries):
{% for var in dir(properties.entries) %}\
${var} \
{% end %}

dir(properties.label):
{% for var in dir(properties.label) %}\
${var} \
{% end %}

dir(properties.name):
{% for var in dir(properties.name) %}\
${var} \
{% end %}

dir(properties.properties):
{% for var in dir(properties.properties) %}\
${var} \
{% end %}
```

When the above file is saved as `/var/lib/bcfg2/TGenshi/test/template.newtxt` and generated with `bcfg2-info buildfile /test test.hostname.org`, the results look like this (below reformatted a little bit to fit in 80 columns):

```
Failed to read file probed.xml
Processed 44 gamin events in 0.108 seconds. 0 collapsed
Processed 17 gamin events in 0.245 seconds. 0 collapsed
Processed 17 gamin events in 0.163 seconds. 0 collapsed
Processed 21 gamin events in 0.197 seconds. 0 collapsed
Processed 0 gamin events in 0.100 seconds. 0 collapsed
Processed 12 gamin events in 0.105 seconds. 0 collapsed
Processed 0 gamin events in 0.100 seconds. 0 collapsed
<?xml version='1.0' encoding='UTF-8'?>
<Path type="file" name="/test" owner="root" perms="644" encoding="ascii" group="root" para
Hostname is test.hostname.org

Groups:
bcfg2-server


Two main ways to get the same property value:
fakeBCFG2password
fakeBCFG2password
```

```
One way to get information about metadata and properties:

dir(metadata):
__class__ __delattr__ __dict__ __doc__ __getattribute__ __hash__ __init__
 __module__ __new__ __reduce__ __reduce_ex__ __repr__ __setattr__ __str__
 __weakref__ all bundles categories get_clients_by_group get_clients_by_profile
 groups hostname inGrouppassword probes uuid

dir(properties):
HandleEvent Index __class__ __delattr__ __dict__ __doc__ __getattribute__
 __hash__ __identifier__ __init__ __iter__ __module__ __new__ __reduce__
 __reduce_ex__ __repr__ __setattr__ __str__ __weakref__ entries label name
 properties

dir(properties.entries):
__add__ __class__ __contains__ __delattr__ __delitem__ __delslice__ __doc__
 __eq__ __ge__ __getattribute__ __getitem__ __getslice__ __gt__ __hash__
 __iadd__ __imul__ __init__ __iter__ __le__ __len__ __lt__ __mul__ __ne__
 __new__ __reduce__ __reduce_ex__ __repr__ __reversed__ __rmul__ __setattr__
 __setitem__ __setslice__ __str__ append count extend index insert pop remove
 reverse sort

dir(properties.label):
__add__ __class__ __contains__ __delattr__ __doc__ __eq__ __ge__
 __getattribute__ __getitem__ __getnewargs__ __getslice__ __gt__ __hash__
 __init__ __le__ __len__ __lt__ __mod__ __mul__ __ne__ __new__ __reduce__
 __reduce_ex__ __repr__ __rmod__ __rmul__ __setattr__ __str__ capitalize center
 count decode encode endswith expandtabs find index isalnum isalpha isdigit
 islower isspace istitle isupper join ljust lower lstrip partition replace
 rfind rindex rjust rpartition rsplit rstrip split splitlinesstartswith strip
 swapcase title translate upper zfill

dir(properties.name):
__add__ __class__ __contains__ __delattr__ __doc__ __eq__ __ge__
 __getattribute__ __getitem__ __getnewargs__ __getslice__ __gt__ __hash__
 __init__ __le__ __len__ __lt__ __mod__ __mul__ __ne__ __new__ __reduce__
 __reduce_ex__ __repr__ __rmod__ __rmul__ __setattr__ __str__ capitalize center
 count decode encode endswith expandtabs find index isalnum isalpha isdigit
 islower isspace istitle isupper join ljust lower lstrip partition replace
 rfind rindex rjust rpartition rsplit rstrip split splitlinesstartswith strip
 swapcase title translate upper zfill

dir(properties.properties):
__class__ __contains__ __copy__ __deepcopy__ __delattr__ __delitem__
 __delslice__ __doc__ __getattribute__ __getitem__ __getslice__ __hash__
 __init__ __iter__ __len__ __new__ __nonzero__ __reduce__ __reduce_ex__
 __repr__ __reversed__ __setattr__ __setitem__ __setslice__ __str__ _init
 addnext addprevious append attrib clear extend find findall findtext get
 getchildren getiterator getnext getparent getprevious getroottree index insert
 items iterancestors iterchildren iterdescendants itersiblings keys makeelement
 nsmap prefix remove replace set sourceline tag tail text values xpath
</Path>
```

Account

The account plugin manages authentication data, including

- /etc/passwd
- /etc/group
- /etc/security/limits.conf
- /etc/sudoers
- /root/.ssh/authorized_keys

User access data is stored in three files in the Account directory:

- superusers (a list of users who always have root privs)
- rootlist (a list of user:host pairs for scoped root privs)
- useraccess (a list of user:host pairs for login access)

SSH keys are stored in files named $username.key; these are installed into root's authorized keys for users in the superusers list as well as for the pertitent users in the rootlike file (for the current system).

Authentication data is read in from (static|dyn).(passwd|group) The static ones are for system local ones, while the dyn. versions are for external synchronization (from ldap/nis/etc). There is also a static.limits.conf that provides the limits.conf header and any static entries.

Cfg

The Cfg plugin provides a repository to describe configuration file contents for clients. In its simplest form, the Cfg repository is just a directory tree modeled off of the directory tree on your client machines.

The Cfg Repository The Cfg plugin is enabled by including **Cfg** on the **plugins** line of the **[server]** section of your Bcfg2 server config file. The repository itself lives in `/var/lib/bcfg2/Cfg`, assuming you are using the default repository location of `/var/lib/bcfg2`. The contents of this directory are a series of directories corresponding to the real-life locations of the files on your clients, starting at the root level. For example:

```
lueningh@tg-prez:~/bcfg2/repository> ls Cfg
bin/  boot/  etc/  opt/  root/  usr/  var/
```

Specific config files go in like-named directories in this heirarchy. For example the password file, `/etc/passwd`, goes in `Cfg/etc/passwd/passwd`, while the ssh pam module config file, `/etc/pam.d/sshd`, goes in `Cfg/etc/pam.d/sshd/sshd`. The reason for the like-name directory is to allow multiple versions of each file to exist, as described below. Note that these files are exact copies of what will appear on the client machine - no templates, XML wrappers, etc.

Group-Specific Files It is often that you want one version of a config file for all of your machines except those in a particular group. For example, `/etc/fstab` should look alike on all of your desktop machines, but should be different on your file servers. Bcfg2 can handle this case through use of group-specific files.

As mentioned above, all Cfg entries live in like-named directories at the end of their directory tree. In the case of fstab, the file at `Cfg/etc/fstab/fstab` will be handed out by default to any client that asks for a copy of `/etc/fstab`. Group-specific files are located in the same directory and are named with the syntax:

```
/path/to/filename/filename.GNN_groupname
```

in which **NN** is a priority number where **00** is lowest and **99** is highest, and **groupname** is the name of a group defined in `Metadata/groups.xml`. Back to our fstab example, we might have a `Cfg/etc/fstab/` directory that looks like:

```
fstab
fstab.G50_server
fstab.G99_fileserver
```

By default, clients will receive the plain fstab file when they request `/etc/fstab`. Any machine that is in the **server** group, however, will instead receive the `fstab.G50_server` file. Finally, any machine that is in the **fileserver** group will receive the `fstab.G99_fileserver` file, even if they are also in the **server** group.

Host-Specific Files Similar to the case with group-specific files, there are cases where a specific machine should have a different version of a file than all others. This can be accomplished with host-specific files. The format of a host-specific file name is:

```
/path/to/filename/filename.H_host.example.com
```

Host-specific files have a higher priority than group specific files. Again, the fstab example:

```
fstab
fstab.G50_server
fstab.G99_fileserver
fstab.H_host.example.com
```

In this case, *host.example.com* will always get the host-specific version, even if it is part of the **server** or **fileserver** (or both) classes.

Note: If you have the ability to choose between using a group-specific and a host-specific file, it is almost always best to use a group-specific one. That way if a hostname changes or an extra copy of a particular client is built, it will get the same changes as the original.

Deltas Bcfg2 has finer grained control over how to deliver configuration files to a host. Let's say we have a Group named file-server. Members of this group need the exact same `/etc/motd` as all other hosts except they need one line added. We could copy motd to `motd.G01_file-server`, add the one line to the Group specific version and be done with it, but we're duplicating data in both files. What happens if we need to update the motd? We'll need to remember to update both files then. Here's where deltas come in. A delta is a small change to the base file. There are two types of deltas: cats and diffs. The cat delta simply

adds or removes lines from the base file. The diff delta is more powerful since it can take a unified diff and apply it to the base configuration file to create the specialized file. Diff deltas should be used very sparingly.

Cat Files Continuing our example for cat files, we would first create a file named `motd.G01_file-server.cat`. The .cat suffix designates that the file is a diff. We would then edit that file and add the following line:

```
+This is a file server
```

The **+** at the begining of the file tells Bcfg2 that the line should be appended to end of the file. You can also start a line with **-** to tell Bcfg2 to remove that exact line wherever it might be in the file. How do we know what base file Bcfg2 will choose to use to apply a delta? The same rules apply as before: Bcfg2 will choose the highest priority, most specific file as the base and then apply deltas in the order of most specific and then increasing in priority. What does this mean in real life. Let's say our machine is a web server, mail server, and file server and we have the following configuration files:

```
motd
motd.G01_web-server
motd.G01_mail-server.cat
motd.G02_file-server.cat
motd.H_foo.example.cat
```

If our machine isn't *foo.example.com* then here's what would happen:

Bcfg2 would choose `motd.G01_web-server` as the base file. It is the most specific base file for this host. Bcfg2 would apply the `motd.G01_mail-server.cat` delta to the `motd.G01_web-server` base file. It is the least specific delta. Bcfg2 would then apply the `motd.G02_file-server.cat` delta to the result of the delta before it. If our machine is foo.example.com then here's what would happen:

Bcfg2 would choose `motd.G01_web-server` as the base file. It is the most specific base file for this host. Bcfg2 would apply the `motd.H_foo.example.com.cat` delta to the `motd.G01_web-server` base file. The reason the other deltas aren't applied to *foo.example.com* is because a **.H_** delta is more specific than a **.G##_** delta. Bcfg2 applies all the deltas at the most specific level.

File permissions File permissions for entries handled by Cfg are controlled via the use of *Info* files. Note that you **cannot** use both a Permissions entry and a Path entry to handle the same file.

Decisions

This page describes the Decisions plugin. The client has support for a centralized set of per-entry installation decisions. This approach is needed when particular changes are deemed "high risk"; this gives the ability to centrally specify these changes, but only install them on clients when administrator supervision is available. Because collaborative configuration is one of the remaining hard issues in configuration management, these issues typically crop up in environments with several administrators and much configuration variety.

In these cases, the client can be configured to run in either a whitelist or blacklist mode, wherein a list of entries is downloaded from the server. The client uses this list to determine which incorrect entries should be corrected during the current run of the installation tool. The Decisions plugin is the only stock plugin that generates entries for client's whitelists or blacklists.

The Decisions plugin uses a directory in the Bcfg2 repository called Decisions. Files in the Decisions subdirectory are named similarly to files managed by Cfg, probes, TCheetah, and TGenshi (so you can use host- and group-specific files and the like after their basename). File basenames are either `whitelist` or `blacklist`. These files have a simple format; the following is an example.

```
$ cat Decisions/whitelist
<Decisions>
  <Decision type='Service' name='*'/>
  <Decision type='Path' name='/etc/apt/apt.conf'/>
</Decisions>
```

Note: To add syntax highlighting in vim, you can add a modeline such as this:

<!– vim: ft=xml –>

This example, included as a whitelist due to its name, enables all services, and the path entry named `/etc/apt/apt.conf`. All these entries must already be present in your repository, the Decisions plugin just references them. In whitelist mode, only the given items are applied to the client; all other entry installation will be surpressed.

In blacklist mode, every entry that is not blacklisted will be installed.

When a client asks for its whitelist or blacklist, all of the files pertaining to that client of the correct type are aggregated into a single list. This list is sent to the client.

Note: This list is only generated when a client is explicitly run with the appropriate option (`-l (whitelist|blacklist)`); client behavior is not controlled unless this option is used. If you do not use Decisions, all your entries will be installed normally.

Note: Also, using this plugin does not present additional prompts or safety nets to the administrator running the client, you have to control these via their respective options (`-I` or `-n`, for example).

Deps

The Deps Plugin allows you to make a series of assertions like "Package X requires Package Y (and optionally also Package Z etc). Note that only configuration entries, like Package, Path, etc can be used. Groupings (like Bundle) are not supported.

Here are some examples:

Note: These particular examples are not extremely useful when using the Packages plugin as Packages will handle the dependency resolution for you. However, there are certainly other use cases for the Deps plugin.

Deps/bcfg2.xml

```
<Dependencies priority='0'>
    <Package name='bcfg2'>
        <Package name='python-lxml'/>
        <Package name='isprelink'/>
    </Package>
</Dependencies>
```

This basically causes any configuration specification that includes Package bcfg2 to include python-lxml and isprelink, in a second base clause.

Deps/bcfg2-server.xml

```
<Dependencies priority='0'>
    <Package name='bcfg2-server'>
        <Package name='python-cheetah'/>
        <Package name='gamin-python'/>
        <Package name='sqlite'/>
        <Package name='python-sqlite'/>
        <Package name='Django'/>
        <Package name='mod_python'/>
        <Package name='graphviz'/>
        <Package name='xorg-x11-font-utils'/>
        <Package name='chkfontpath'/>
        <Package name='ttmkfdir'/>
        <Package name='xorg-x11-xfs'/>
        <Package name='urw-fonts'/>
    </Package>
</Dependencies>
```

This states that the bcfg2-server package (it's a separate package on some distros) depends on a long list of other packages.

Hostbase

IP management system built on top of Bcfg2. It has four main parts: a django data model, a web frontend, command-line utilities, and a Bcfg2 plugin that generates dhcp, dns, and yp configuration files.

Installation Installation of Hostbase requires installation of a python module, configuration of database (mysql or postgres), and configuration of an Apache webserver with mod_python. Hostbase was developed using MySQL, so this document is aimed at MySQL users.

Prerequisites

- mysql
- python-mysqldb
- Django

Configure the database Create the hostbase database and a user. For MySQL users:

```
mysql> CREATE DATABASE hostbase
mysql> quit

systemprompt#:  mysql -u root hostbase
mysql> GRANT ALL PRIVILEGES ON *.* TO hostbaseuser@mycomputer.private.net IDENTIFIED
```

```
       BY 'password' WITH GRANT OPTION;
mysql> quit
```

As of Bcfg2 v0.8.7 configuration options for Hostbase have moved to `/etc/bcfg2.conf`. There is an example bcfg2.conf with Hostbase options located at `bcfg2-tarball/examples/bcfg2.confHostbase`. Edit the hostbase options to correspond to the database you've initialized and copy the configuration to `/etc/bcfg2.conf`. To finish creating the database, from your `path to python/Bcfg2/Server/Hostbase` directory, run `python manage.py syncdb` to do all table creation.

Configure the web interface Now it's possible to explore the Hostbase web interface. For curiosity, you can run Django's built-in development server to take a peek. Do this by running `python manage.py runserver [servername:port]` from your Hostbase directory. Django will default to `localhost:8000` if no server or port is entered. Now you can explore the web interface. Try adding a host and a zone. You'll see that a ".rev" zone already exists. This is where information for reverse files will go.

For production, you'll want to have this configured for Apache with mod_python. Here is an example of how to configure Hostbase as a virtual host.

```
<VirtualHost hostbase.mcs.anl.gov:80>
        ServerAdmin systems@mcs.anl.gov

        DocumentRoot /var/www/hostbase/
        <Directory />
                AllowOverride None
        </Directory>

        # Possible values include: debug, info, notice, warn, error, crit,
        # alert, emerg.
        LogLevel warn

        ServerSignature Off

        # Stop TRACE/TRACK vulnerability
        <IfModule mod_rewrite.c>
                RewriteEngine on
                RewriteCond %{REQUEST_METHOD} ^(TRACE|TRACK)
                RewriteRule .* - [F]
        </IfModule>

        Redirect / https://hostbase.mcs.anl.gov/
</VirtualHost>

<VirtualHost hostbase.mcs.anl.gov:443>
        ServerAdmin systems@mcs.anl.gov

        DocumentRoot /var/www/hostbase/
        <Directory />
                AllowOverride None
        </Directory>
```

```
        # Possible values include: debug, info, notice, warn, error, crit,
        # alert, emerg.
        LogLevel warn

        ServerSignature Off

        # Stop TRACE/TRACK vulnerability
        <IfModule mod_rewrite.c>
                RewriteEngine on
                RewriteCond %{REQUEST_METHOD} ^(TRACE|TRACK)
                RewriteRule .* - [F]
        </IfModule>

        SSLEngine On
        SSLCertificateFile /etc/apache2/ssl/hostbase_server.crt
        SSLCertificateKeyfile /etc/apache2/ssl/hostbase_server.key

<Location "/">
        SetHandler python-program
        PythonHandler django.core.handlers.modpython
        SetEnv DJANGO_SETTINGS_MODULE Bcfg2.Server.Hostbase.settings
        PythonDebug On
</Location>
<Location "/site_media/">
        SetHandler None
</Location>
</VirtualHost>
```

You'll need to copy the contents of `Hostbase/media` into `/var/www/hostbase/site_media` in this configuration to serve the correct css files.

Enable the Hostbase plugin Now that the database is accessible and there is some data in it, you can enable the Hostbase plugin on your Bcfg2 server to start generating some configuration files. All that needs to be done is to add `Hostbase` to the end of the list of generators in your bcfg2.conf file. To see what's being generated by Hostbase, fire up a Bcfg2 development server: `bcfg2-info`. For more information on how to use the Bcfg2 development server, type help at the prompt. For our purposes, type `debug`. This will bring you to an interactive python prompt where you can access bcfg's core data.

```
for each in bcore.plugins['Hostbase'].filedata:
    print each
```

The above loop will print out the name of each file that was generated by Hostbase. You can see the contents of any of these by typing `print bcore.plugins['Hostbase'].filedata[filename]`.

Create a bundle Bcfg2 needs a way to distribute the files generated by Hostbase. We'll do this with a bundle. In bcfg's `Bundler` directory, touch `hostbase.xml`.

```
<Bundle name='hostbase' version='0.1'>
   <Package name='dhcp3-server'/>
   <Package name='bind9'/>
```

```
    <Service name='dhcp3-server'/>
    <Service name='bind9'/>
    <Path name='/etc/dhcp3/dhcpd.conf'/>
    <Path name='/etc/bind/[your domain]'/>
    <Path name='/etc/bind/xxx.xxx.xxx.rev'/>
</Bundle>
```

The above example is a bundle that will deliver both dhcp and dns files. This can be trivially split into separate bundles. It is planned that Hostbase will eventually be able to generate the list of `Paths` in its bundles automatically.

Do a Hostbase push You'll want to be able to trigger the Hostbase plugin to rebuild it's config files and push them out when data has been modified in the database. This can be done through and XMLRPC function available from the Bcfg2 server. From a client that is configured to receive one or more hostbase bundles, you'll need to first edit your `python/site-packages/Bcfg2/Client/Proxy.py` file. Add `'Hostbase.rebuildState'` to the list of methods in the Bcfg2 client proxy object. The modified list is shown below:

```
class bcfg2(ComponentProxy):
    '''bcfg2 client code'''
    name = 'bcfg2'
    methods = ['AssertProfile', 'GetConfig', 'GetProbes', 'RecvProbeData', 'RecvStats', 'Ho
```

Now copy the file `hostbasepush.py` from `bcfg2/tools` in the Bcfg2 source to your machine. When this command is run as root, it triggers the Hostbase to rebuild it's files, then runs the Bcfg2 client on your local machine to grab the new configs.

NIS Authentication Django allows for custom authentication backends to its login procedure. Hostbase has an NIS authentication backend that verifies a user to be in the unix group allowed to modify Hostbase.

To enable this feature:

- first edit your `Hostbase/settings.py` file and uncomment the line **Hostbase.backends.NISBackend** in the list of *AUTHENTICATION_BACKENDS*
- enter the name of the unix group you want to give access to Hostbase in the *AUTHORIZED_GROUP* variable
- in your `Hostbase/hostbase/views.py` file at the very bottom, uncomment the block(s) of lines that give you the desired level of access

Hostbase will now direct the user to a login page if he or she is not authorized to view a certain page. Users should log in with their regular Unix username and password.

NagiosGen

This page describes the installation and use of the NagiosGen plugin.

Update `/etc/bcfg2.conf`, adding NagiosGen to plugins:

```
plugins = SSHbase,Cfg,Pkgmgr,Rules,TCheetah,TWbase,NagiosGen
```

Create the NagiosGen directory:

```
$ mkdir /var/lib/bcfg2/NagiosGen
```

Create default host, and group specs in:

- /var/lib/bcfg2/NagiosGen/default-host.cfg:

```
define host{
        name                            default
        check_command                   check-host-alive
        check_interval                  5
        check_period                    24x7
        contact_groups                  admins
        event_handler_enabled           1
        failure_prediction_enabled      1
        flap_detection_enabled          1
        initial_state                   o
        max_check_attempts              10
        notification_interval           0
        notification_options            d,u,r
        notification_period             workhours
        notifications_enabled           1
        process_perf_data               0
        register                        0
        retain_nonstatus_information    1
        retain_status_information       1
        retry_interval                  1
        }
```

- /var/lib/bcfg2/NagiosGen/default-group.cfg:

```
define service{
        name                            default-service
        active_checks_enabled           1
        passive_checks_enabled          1
        obsess_over_service             0
        check_freshness                 0
        notifications_enabled           1
        event_handler_enabled           1
        flap_detection_enabled          1
        process_perf_data               0
        retain_status_information       1
        retain_nonstatus_information    1
        is_volatile                     0

        check_period                    24x7
        max_check_attempts              4
        check_interval                  5
        retry_interval                  1
        contact_groups                  admins
        notification_options            w,u,c,r
```

```
        notification_interval           0
        notification_period             workhours
        }
```

Create group configuration files (Named identical to Bcfg2 groups) and add services, and commands specific to the hostgroup (Bcfg2 group) in

- /var/lib/bcfg2/NagiosGen/base-group.cfg:

```
define hostgroup{
        hostgroup_name  base
        alias           base
        notes           Notes
}

define service{
        service_description             NTP
        check_command                   check_ntp!
        use                             default-service
        hostgroup_name                  base
        }

define command{
        command_name    check_ssh
        command_line    $USER1$/check_ssh $ARG1$ $HOSTADDRESS$
        }

define service{
        service_description             SSH
        check_command                   check_ssh!
        use                             default-service
        hostgroup_name                  base
        }
```

- /var/lib/bcfg2/NagiosGen/web-server-group.cfg:

```
define hostgroup{
        hostgroup_name  web-server
        alias           Port 80 Web Servers
        notes           UC/ANL Teragrid Web Servers Running on Port 80
}

define command{
        command_name    check_http_80
        command_line    $USER1$/check_http $HOSTADDRESS$
        }

define service{
        service_description             HTTP:80
        check_command                   check_http_80!
        use                             default-service
        hostgroup_name                  web-server
        }
```

Create a nagios Bcfg2 bundle `/var/lib/bcfg2/Bundler/nagios.xml`

```
<Bundle name='nagios' version='2.0'>
  <Path name='/etc/nagiosgen.status'/>
  <Group name='rh'>
    <Group name='nagios-server'>
      <Path name='/etc/nagios/nagiosgen.cfg'/>
      <Package name='libtool-libs'/>
      <Package name='nagios'/>
      <Package name='nagios-www'/>
      <Service name='nagios'/>
    </Group>
  </Group>
  <Group name='debian-lenny'>
    <Group name='nagios-server'>
      <Path name='/etc/nagios3/nagiosgen.cfg'
                altsrc='/etc/nagios/nagiosgen.cfg'/>
      <Package name='nagios3'/>
      <Package name='nagios3-common'/>
      <Package name='nagios3-doc'/>
      <Package name='nagios-images'/>
      <Service name='nagios3'/>
    </Group>
  </Group>
</Bundle>
```

Assign clients to nagios groups in `/var/lib/bcfg2/Metadata/groups.xml`

```
<Group name='nagios'>
  <Bundle name='nagios'/>
</Group>

<Group name='nagios-server'>
  <Bundle name='nagios'/>
</Group>
```

Update nagios configuration file to use `nagiosgen.cfg`:

```
cfg_file=/etc/nagios/nagiosgen.cfg
```

Note that some of these files are built on demand, each time a client in group "nagios-server" checks in with the Bcfg2 server. Local nagios instances can be configured to use the NagiosGen directory in the Bcfg2 repository directly.

Packages

New in version 1.0.0. This page documents the Packages plugin. Packages is an alternative to *Pkgmgr* for specifying package entries for clients. Where Pkgmgr explicitly specifies package entry information, Packages delegates control of package version information to the underlying package manager, installing the latest version available through those channels.

"Magic Groups" Packages is the only plugin that uses "magic groups". Most plugins operate based on client group memberships, without any concern for the particular names chosen for groups by the user. The Packages plugin is the sole exception to this rule. Packages needs to "know" two different sorts of facts about clients. The first is the basic OS/distro of the client, enabling classes of sources. The second is the architecture of the client, enabling sources for a given architecture. In addition to these magic groups, each source may also specify a non-magic group to limit the source's applicability to group member clients.

Source	OS Group	Architecture
APTSource	debian	i386
APTSource	ubuntu	amd64
APTSource	nexenta	
YUMSource	redhat	i386
YUMSource	centos	x86_64
YUMSource	fedora	x86_64

Limiting sources to groups Each source can also specify explicit group memberships. In the following example, the ubuntu-hardy group is also required. Finally, clients must be a member of the appropriate architecture group, in this case, either i386 or amd64. In total, in order for this source to be associated with a client is for the client to be in one of the sentinel groups (debian, ubuntu, or nexenta), the explicit group ubuntu-hardy, and any of the architecture groups (i386 or amd64).

Memberships in architecture groups is needed so that Packages can map software sources to clients. There is no other way to handle this than to impose membership in the appropriate architecture group.

When multiple sources are specified, clients are associated with each source to which they apply (based on group memberships, as described above). Packages and dependencies are resolved from all applicable sources.

Note: To recap, a client needs to be a member of the **OS Group**, **Architecture** group, and any other groups defined in your `Packages/config.xml` file in order for the client to be associated to the proper sources.

Setup Three basic steps are required for Packages to work properly.

1. Create Packages/config.xml. This file should look approximately like the example below, and describes both which software repositories should be used, and which clients are eligible to use each one.
2. Ensure that clients are members of the proper groups. Each client should be a member of one of the sentinel groups listed above (debian/ubuntu/redhat/centos/nexenta), all of the groups listed in the source (like ubuntu-intrepid or centos-5.2 in the following examples), and one of the architecture groups listed in the source configuration (i386, amd64 or x86_64 in the following examples). ''Failure to do this will result in the source either not applying to the client, or only architecture independent packages being made available to the client.''
3. Add Package entries to bundles.
4. Sit back and relax, as dependencies are resolved, and automatically added to client configurations.

Prerequisite Resolution Packages provides a prerequisite resolution mechanism which has no analogue in Pkgmgr. During configuration generation, all structures are processed. After this phase, but before entry binding, a list of packages and the client metadata instance is passed into Packages' resolver. This process determines a superset of packages that will fully satisfy dependencies of all package entries included in structures, and reports any prerequisites that cannot be satisfied. This facility should largely remove the need to use the *Base* plugin.

Disabling dependency resolution New in version 1.1.0. Dependency resolution can now be disabled by adding this to Sources in config.xml:

```
<Sources>
    <Config resolver="disabled" />
    ...
</Sources>
```

All metadata processing can be disabled as well:

```
<Sources>
    <Config metadata="disabled" />
    ...
</Sources>
```

Example usage Create a config.xml file in the Packages directory that looks something like this:

```
<Sources>
  <APTSource>
    <Group>ubuntu-intrepid</Group>
    <URL>http://us.archive.ubuntu.com/ubuntu</URL>
    <Version>intrepid</Version>
    <Component>main</Component>
    <Component>universe</Component>
    <Arch>i386</Arch>
    <Arch>amd64</Arch>
  </APTSource>
</Sources>
```

Note: The default behavior of the Packages plugin is to not make any assumptions about which packages you want to have added automatically. For that reason, neither **Recommended** nor **Suggested** packages are added as dependencies by default. You will notice that the default behavior for apt is to add Recommended packages as dependencies. You can configure the Packages plugin to add recommended packages by adding the following: New in version 1.1.0.

```
<Recommended>True</Recommended>
```

Yum sources can be similarly specified:

```
<Sources>
  <YUMSource>
    <Group>centos-5.2</Group>
    <URL>http://mirror.centos.org/centos/</URL>
    <Version>5.2</Version>
    <Component>os</Component>
```

```
    <Component>updates</Component>
    <Component>extras</Component>
    <Arch>i386</Arch>
    <Arch>x86_64</Arch>
  </YUMSource>
</Sources>
```

Note: There is also a RawURL syntax for specifying APT or YUM sources that don't follow the conventional layout:

```
<Sources>
  <!-- CentOS (5.4) sources -->
  <YUMSource>
    <Group>centos5.4</Group>
    <RawURL>http://mrepo.ices.utexas.edu/centos5-x86_64/RPMS.os</RawURL>
    <Arch>x86_64</Arch>
  </YUMSource>
  <YUMSource>
    <Group>centos5.4</Group>
    <RawURL>http://mrepo.ices.utexas.edu/centos5-x86_64/RPMS.updates</RawURL>
    <Arch>x86_64</Arch>
  </YUMSource>
  <YUMSource>
    <Group>centos5.4</Group>
    <RawURL>http://mrepo.ices.utexas.edu/centos5-x86_64/RPMS.extras</RawURL>
    <Arch>x86_64</Arch>
  </YUMSource>
</Sources>
```

```
<Sources>
  <APTSource>
    <Group>ubuntu-lucid</Group>
    <RawURL>http://hudson-ci.org/debian/binary</RawURL>
    <Arch>amd64</Arch>
  </APTSource>
  <APTSource>
    <Group>ubuntu-lucid</Group>
    <RawURL>http://hudson-ci.org/debian/binary</RawURL>
    <Arch>i386</Arch>
  </APTSource>
</Sources>
```

Configuration Updates Packages will reload its configuration upon an explicit command via bcfg2-admin.:

```
[0:3711] bcfg2-admin xcmd Packages.Refresh
True
```

During this command (which will take some time depending on the quantity and size of the sources listed in the configuration file), the server will report information like:

```
Packages: Updating http://mirror.anl.gov/ubuntu//dists/jaunty/main/binary-i386/Packages.gz
Packages: Updating http://mirror.anl.gov/ubuntu//dists/jaunty/main/binary-amd64/Packages.gz
Packages: Updating http://mirror.anl.gov/ubuntu//dists/jaunty/universe/binary-i386/Packages
Packages: Updating http://mirror.anl.gov/ubuntu//dists/jaunty/universe/binary-amd64/Package
...
Packages: Updating http://mirror.centos.org/centos/5/extras/x86_64/repodata/filelists.xml.g
Packages: Updating http://mirror.centos.org/centos/5/extras/x86_64/repodata/primary.xml.gz
```

Once line per file download needed. Packages/config.xml will be reloaded at this time, so any source specification changes (new or modified sources in this file) will be reflected by the server at this point.

Soft reload New in version 1.1.0. A soft reload can be performed to reread the configuration file and download only missing sources.:

```
[0:3711] bcfg2-admin xcmd Packages.Reload
True
```

Availability Support for clients using yum and apt is currently available. Support for other package managers (Portage, Zypper, IPS, etc) remain to be added.

Validation A schema for Packages/config.xml is included; config.xml can be validated using `bcfg2-repo-validate`.

Note: The schema requires that elements be specified in the above order.

Limitations Packages does not do traditional caching as other plugins do. Changes to the Packages config file require a server restart for the time being.

Package Verification In order to do disable per-package verification Pkgmgr style, you will need to use *BoundEntries* like below

```
<BoundPackage name="mem-agent" priority="1" version="auto" type="yum" verify="false"/>
```

Generating Client APT/Yum Configurations New in version 1.1.0. Client repository information can be generated automatically from software sources using *TGenshi* or *TCheetah*. A list of source urls are exposed in the client's metadata as metadata.Packages.sources.

An example *TGenshi* APT template:

```
# bcfg2 maintained apt

{% for s in metadata.Packages.sources %}\
deb ${s.url}${s.version} ${s.groups[0]} {% for comp in s.components %}$comp {% end %}

{% end %}\
```

An example *TGenshi* YUM template:

```
# bcfg2 maintained yum

% for s in metadata.Packages.sources %}\
[${s.groups[0]}_${s.component}]
name=${s.groups[0]}_${s.component}
baseurl=${s.url}

{% end %}\
```

Debugging unexpected behavior

Using bcfg2-info The dependency resolver used in Packages can be run in debug mode:

```
$ bcfg2-info
...
Handled 20 events in 0.004s
> debug
dropping to python interpreter; press ^D to resume
...
(debug_shell)
>>> m = self.build_metadata('ubik3')
>>> self.plugins['Packages'].complete(m, ['ssh'], debug=True)
Package ssh: adding new deps ['openssh-client', 'openssh-server']
Package openssh-server: adding new deps ['libc6', 'libcomerr2', 'libkrb53', 'libpam0g', 'li
', 'libwrap0', 'zlib1g', 'debconf', 'libpam-runtime', 'libpam-modules', 'adduser', 'dpkg',
Package debconf: adding new deps ['debconf-i18n']
Package libpam-modules: adding new deps ['libdb4.7']
Package openssh-client: adding new deps ['libedit2', 'libncurses5', 'passwd']
Package lsb-base: adding new deps ['sed', 'ncurses-bin']
Package adduser: adding new deps ['perl-base']
Package debconf-i18n: adding new deps ['liblocale-gettext-perl', 'libtext-iconv-perl', 'lib
Package passwd: adding new deps ['debianutils']
Package libtext-charwidth-perl: adding new deps ['perlapi-5.10.0']
VPackage perlapi-5.10.0: got provides ['perl-base']
Package libkrb53: adding new deps ['libkeyutils1']
Package libtext-iconv-perl: adding new deps ['perlapi-5.10.0']
Package libc6: adding new deps ['libgcc1', 'findutils']
Package libgcc1: adding new deps ['gcc-4.3-base']
(set(['debconf', 'libgcc1', 'lsb-base', 'libtext-wrapi18n-perl', 'libtext-iconv-perl', 'sed
```

This will show why the resolver is acting as it is. Replace "ubik3" and ['ssh'] with a client name and list of packages, respectively. Also, a more polished interface to this functionality is coming as well.

Each line starting with Package: <name> describes a set of new prerequisites pulled in by a package. Lines starting with VPackage <vname> describe provides entries and their mappings to required names. The last line describes the overall results of the resolver, with three fields: a list of packages that should be installed, a list of unresolved requirements, and a type for these packages.

Using bcfg2-server Once the server is started, enable debugging via bcfg2-admin:

```
$ bcfg2-admin xcmd Packages.toggle_debug
```

TODO list

- Zypper support
- Portage support
- Explicit version pinning (a la Pkgmgr)

Developing for Packages In order to support a given client package tool driver, that driver must support use of the auto value for the version attribute in Package entries. In this case, the tool driver views the current state of available packages, and uses the underlying package manager's choice of correct package version in lieu of an explicit, centrally-specified, version. This support enables Packages to provide a list of Package entries with version='auto'. Currently, the APT and YUMng drivers support this feature. Note that package management systems without any network support cannot operate in this fashion, so RPMng and SYSV will never be able to use Packages. Emerge, Zypper, IPS, and Blastwave all have the needed features to be supported by Packages, but support has not yet been written.

Packages fills two major functions in configuration generation. The first is to provide entry level binding support for Package entries included in client configurations. This function is quite easy to implement; Packages determines (based on client group membership) if the package is available for the client system, and which type it has. Because version='auto' is used, no version determination needs to be done.

The second major function is more complex. Packages ensures that client configurations include all package-level prerequisites for package entries explicitly included in the configuration. In order to support this, Packages needs to directly process network data for package management systems (the network sources for apt or yum, for examples), process these files, and build data structures describing prerequisites and the providers of those functions/paths. To simplify implementations of this, there is a generic base class (Bcfg2.Server.Plugins.Packages.Source) that provides a framework for fetching network data via HTTP, processing those sources (with subclass defined methods for processing the specific format provided by the tool), a generic dependency resolution method, and a caching mechanism that greatly speeds up server/bcfg2-info startup.

Each source type must define:

- a get_urls attribute (and associated urls property) that describes the URLS where to get data from.
- a read_files method that reads and processes the downloaded files

Sources may define a get_provides method, if provides are complex. For example, provides in rpm can be either rpm names or file paths, so multiple data sources need to be multiplexed.

The APT source in `src/lib/Server/Plugins/Packages.py` provides a relatively simple implementation of a source.

Pkgmgr

Note: See [wiki:ClientTools/RPMng#PackageTagNewStyleandAttributes RPMng#PackageTagNewStyleandAttributes].'' The way of showing the architecture of the RPM has

changed. The new way is “arch”. The old way is “multiarch”. ‘’‘This document needs to be updated and include version of Bcfg2 where change took place.’‘’

The Pkgmgr plugin resolves the Abstract Configuration Entity “Package” to a package specification that the client can use to detect, verify and install the specified package.

For a package specification to be included in the Literal configuration the name attribute from an Abstract Package Tag (from Base or Bundler) must match the name attribute of a Package tag in Pkgmgr, along with the appropriate group associations of course.

Each file in the Pkgmgr directory has a priority. This allows the same package to be served by multiple files. The priorities can be used to break ties in the case that multiple files serve data for the same package.

Usage of Groups in Pkgmgr Groups are used by the Pkgmgr plugin, along with host metadata, for selecting the package entries to include in the clients literal configuration. They can be thought of as:

```
if client is a member of group1 then
    assign to literal config
```

Nested groups are conjunctive (logical and).:

```
if client is a member of group1 and group2 then
    assign to literal config
```

Group membership may be negated.

Tag Attributes in Pkgmgr

PackageList Tag The PackageList Tag may have the following attributes:

Name	Description	Values
priority	Sets the priority for packages in the package list. The higher value wins.	Integer
type	Package type that applies to all packages in the list. This value is inherited by all packages without an explicit type attribute.	deb|rpm|blast|encap|sysv|portage|yum
uri	URI to prepend to filename sto fetch packages in this list.	String
multiarch	Comma-separated list of architectures that apply to all packages in this list. Inherited by all package entries in the file that does not have this attribute explicitly.	String
srcs	To be used with multiarch support. Inherited by all Package entries without this attribute	String

Pkgmgr Group Tag The Pkgmgr Group Tag may have the following attributes:

Name	Description	Values
name	Group Name	String
negate	Negate group membership (is not a member of)	True|False

Package Tag The Package Tag may have the following attributes:

Name	Description	Values
name	Package Name	String
version	Package version, set to `auto` to install the latest version in the client's cache, or `any` to verify that any version of the package is installed on the client	String
file	Package file name. Several other attributes (name, version) can be automatically defined based on regular expressions defined in the Pkgmgr plugin.	String
simple-file	Package file name. No name parsing is performed, so no extra fields get set	String
verify	Whether package verification should be done	True|False
multiarch	Comma-separated list of the architectures of this package that should be installed. (Temporary work-around)	String
srcs	File name creation rules for multiarch packages. (Temporary work-around)	String
type	Package type (rpm, yum, apt, sysv, blast)	String

Client Tag The Client Tag is used in a PackageList for selecting the package entries to include in the clients literal configuration. Its function is similar to the Group tag in this context. It can be thought of as:

```
if client is name then
    assign to literal config
```

The Client Tag may have the following attributes:

Name	Description	Values
name	Client Name	String
negate	Negate client selection (if not client name)	True|False

Pkgmgr Directory The Pkgmgr/ directory keeps the XML files that define what packages are available for a host or image and where to find those packages. All the files in the directory are processed.

The names of the XML files have no special meaning to Bcfg2; they are simply named so it's easy for the administrator to know what the contents hold. All Packages could be kept in a single file if so desired. Bcfg2 simply uses the Groups in the files and priorities to determine how to assign Packages to a host's literal configuration.

Listed detailed below is one possible structure for the Pkgmgr directory.

The files are structured to contain particular portions of distribution repositories.

The files in the directory are:

```
$ ls Pkgmgr/
centos-4-noarch-updates.xml
centos-4-x86_64-updates.xml
centos-4-x86_64.xml
backup.example.com.xml
fedora-core-4-noarch-updates.xml
```

```
fedora-core-4-x86-updates.xml
fedora-core-4-x86.xml
rhel-as-4-noarch-updates.xml
rhel-as-4-x86-updates.xml
rhel-as-4-x86.xml
rhel-es-4-noarch-updates.xml
rhel-es-4-x86-updates.xml
rhel-es-4-x86.xml
rhel-ws-4-noarch-udpates.xml
rhel-ws-4-x86_64-updates.xml
rhel-ws-4-x86_64.xml
rhel-ws-4-x86-updates.xml
rhel-ws-4-x86.xml
```

As can be seen the file names have been selected to indicate what the contents are and have been split by Vendor, product and repository area.

A partial listing of the centos-4-x86_64.xml is below

```
$ cat centos-4-x86_64.xml
<PackageList uri='http://www.example.com/yam/Centos44-x86_64/RPMS.os/' type='yum' priority=
    <Group name='Centos4.4-Standard'>
        <Group name='x86_64'>
            <Package name='audit-libs' version='1.0.13-1.EL4' type='yum'/>
            <Package name='audit' version='1.0.13-1.EL4' type='yum'/>
            <Package name='basesystem' version='8.0-2' type='yum'/>
            <Package name='bash' version='3.0-18.1' type='yum'/>
            <Package name='bcfg2' version='0.9.1-0.1' type='yum'/>
            <Package name='beecrypt' version='3.1.0-3' type='yum'/>
            ...
            <Package name='VMwareTools' version='6530-29996' type='yum'/>
            <Package name='yum' version='2.4.2-1' type='yum'/>
            <Package name='zlib' version='1.2.1.2-1.2' type='yum'/>
        </Group>
  </Group>
</PackageList>

$ cat centos-4-x86_64-updates.xml
<PackageList uri='http://www.example.com/yam/Centos44-x86_64/RPMS.updates/' type='yum' pri
    <Group name='Centos4.4-Standard'>
        <Group name='x86_64'>
            <Package name='audit-libs' version='1.0.14-1.EL4' type='yum'/>
            <Package name='audit' version='1.0.14-1.EL4' type='yum'/>
            <Package name='basesystem' version='8.0-4' type='yum'/>
            <Package name='bash' version='3.0-19.3' type='yum'/>
            <Package name='bcfg2' version='0.9.2-0.1' type='yum'/>
            <Package name='beecrypt' version='3.1.0-6' type='yum'/>
            ...
            <Package name='VMwareTools' version='6530-29996' type='yum'/>
            <Package name='yum' version='2.4.3-1' type='yum'/>
            <Package name='zlib' version='1.2.1.2-1.2' type='yum'/>
        </Group>
  </Group>
</PackageList>
```

Here it can be seen that the data is encapsulated in a !PackageList Tag which describes the URI of the files described, the type of package, and the priority of the files in this list.

The priority is used to decide which specific file to use when there are multiple files that could be used for a particular host. The highest priority file is the one that is used.

Using this system, it is possible to have a file that contains all the Packages from the original installation, centos-4-x86_64.xml in this case, and then create a new file that contains updates that are made available afterwards, centos-4-x86_64-updates.xml and centos-4-noarch-updates.xml in this case. The priority of the update PackageLists just needs to be higher so that they will be selected instead of the original installation Packages.

The backup.example.com.xml contains a packalist for a specific host which is qualified by the Client tag. Its Packages have a higher priority than the update Packages. This is because this particular host requires special Packages that are older than the ones available in the updates.

```
<PackageList uri='http://www.example.com/yam/Centos44-x86_64/RPMS.os/' type='yum' priority=
    <Client name='server86.example.com'>
        <Package name='audit-libs' version='1.0.13-1.EL4' type='yum'/>
        <Package name='audit' version='1.0.13-1.EL4' type='yum'/>
        <Package name='basesystem' version='8.0-2' type='yum'/>
        <Package name='bash' version='3.0-18.1' type='yum'/>
        <Package name='bcfg2' version='0.9.1-0.1' type='yum'/>
        <Package name='beecrypt' version='3.1.0-3' type='yum'/>
        ...
        <Package name='VMwareTools' version='6530-29996' type='yum'/>
        <Package name='yum' version='2.4.2-1' type='yum'/>
        <Package name='zlib' version='1.2.1.2-1.2' type='yum'/>
    </Client>
</PackageList>
```

Simplifying Multi-Architecture Environments with *Altsrc* Frequently multi-architecture environments (typically x86_64) will run into problems needing to specify different architectures on different groups for clients. For example, desktop machines may install 32-bit compatibility packages in addition to 64-bit ones, while servers may install only 64-bit packages. Specifying this in the Pkgmgr was onerous, because different package targets (64bit, 32+64, etc) needed to be specified on a package by group basis. Two features have been implemented that should ease this situation considerably.

- The *Altsrc* feature adds the ability to add a "bind as" directive to entries. For example, the following entry, in a bundle:

```
  <Package name='foo' altsrc='bar'/>

would bind as if it were named bar, while the entry would still appear named "foo" in the c
```

- Pkgmgr now builds virtual package targets for any package with Instance client elements. This means that if a client attempts to bind:

```
  <Package name='libfoo:x86_64,i686'/>
It will only include the instances listed in the package.
By using these features together, a bundle can include:
```

```
  <Package name='libfoo' altsrc='libfoo:i386,i686'/>
```

This in conjunction with a Pkgmgr entry that looks like:

```
  <Package name='libfoo'>
    <Instance arch='i386' version='1.0.4-12'/>
    <Instance arch='i586' version='1.0.4-12'/>
    <Instance arch='i686' version='1.0.4-12'/>
    <Instance arch='x86_64' version='1.0.4-12'/>
  </Package>
```

Will result in a bound entry that looks like:

```
<Package name='libfoo'>
  <Instance arch='i386' version='1.0.4-12'/>
  <Instance arch='i686' version='1.0.4-12'/>
</Package>
```

Altogether, this should move policy decisions about package architectures to bundles/base.

Client Driver (Plugins) Specific Attributes Not all the attributes that are available in Pkgmgr are honoured by all the client drivers. The following client drivers (plugins) have driver specific attributes:

- *RPMng*

Rules

The Rules plugin resolves the following Abstract Configuration Entities:

- Service
- Package
- Path
- Action

to literal configuration entries suitable for the client drivers to consume.

For an entity specification to be included in the Literal configuration the name attribute from an Abstract Entity Tag (from Base or Bundler) must match the name attribute of an Entity tag in Rules, along with the appropriate group associations of course.

Each file in the Rules directory has a priority. This allows the same Entities to be served by multiple files. The priorities can be used to break ties in the case that multiple files serve data for the same Entity.

Usage of Groups in Rules Groups are used by the Rules plugin, along with host metadata, for selecting the Configuration Entity entries to include in the clients literal configuration. They can be thought of as:

```
if client is a member of group1 then
    assign to literal config
```

Nested groups are conjunctive (logical and).:

```
if client is a member of group1 and group2 then
    assign to literal config
```

Group membership may be negated.

Tag Attributes in Rules

Rules Tag The Rules Tag may have the following attributes:

Name	Description	Values
priority	Sets the priority for Rules in this Rules list.The higher value wins.	String

Rules Group Tag The Rules Group Tag may have the following attributes:

Name	Description	Values
name	Group Name	String
negate	Negate group membership (is not a member of)	(True|False)

Package Tag The Package Tag may have the following attributes:

Name	Description	Values
name	Package Name	String
version	Package Version or version='noverify' to not do version checking in the Yum driver only (temporary work a round).	String
file	Package file name. Several other attributes (name, version) can be automatically defined based on regular expressions defined in the Pkgmgr plugin.	String
simple-file	Package file name. No name parsing is performed, so no extra fields get set	String
verify	verify='false' - do not do package verification	String
reloc	RPM relocation path.	String
multiarch	Comma separated list of the architectures of this package that should be installed.	String
srcs	Filename creation rules for multiarch packages.	String
type	Package type. (rpm, yum, apt,sysv,blast)	String

Action Tag See *Actions*

Service Tag

Name	Description	Values
mode	Per Service Mode (New in 1.0)	(manual|default|supervised|custom)
name	Service Name	String
status	Should the service be on or off (default: off).	(on|off)
target	Service command for restart, modified targets require mode="custom" (default: restart)	String
type	Driver to use on the client to manage this service.	(chkconfig|deb|rc-update|smf|upstart)
sequence	Order for service startup (debian services only)	integer

Service mode descriptions New in version 1.0.0.

- manual
  - do not start/stop/restart this service
- default
  - perform appropriate service operations
- supervised
  - default and ensure service is running (or stopped) when verification is performed
  - deprecates supervised='true'
- custom
  - set non-default service command for restart (use in conjunction with target attribute)

Client Tag The Client Tag is used in Rules for selecting the package entries to include in the clients literal configuration. Its function is similar to the Group tag in this context. It can be thought of as:

```
if client is name then
    assign to literal config
```

The Client Tag may have the following attributes:

Name	Description	Values
name	Client Name	String
negate	Negate client selection (if not client name)	(True|False)

Path Tag The Path tag has different values depending on the *type* attribute of the path specified in your configuration. Below is a set of tables which describe the attributes available for various Path types.

device

Name	Description	Values
name	Name of the device	String
dev_type	Type of device	(block|char|fifo)
owner	Device owner	String
group	Device group	String
major	Major number (block or char devices)	integer
minor	Minor number (block or char devices)	integer

directory

Name	Description	Values
name	Directory Name	String
perms	Permissions of the directory	String
owner	Owner of the directory	String
group	Group Owner of the directory	String
prune	prune unspecified entries from the Directory	true|false

hardlink

Name	Description	Values
name	Name of the hardlink	String
to	File to link to	String

nonexistent

Name	Description	Values
name	Name of the (nonexistent) file	String
type	Type of file	nonexistent

permissions

Name	Description	Values
name	Name of the file.	String
perms	Permissions of the file.	String
owner	Owner of the file.	String
group	Group of the file.	String

symlink

Name	Description	Values
name	Name of the symlink.	String
to	File to link to	String

Rules Directory The Rules/ directory keeps the XML files that define what rules are available for a host. All the files in the directory are processed.

The names of the XML files have no special meaning to Bcfg2; they are simply named so it's easy for the administrator to know what the contents hold. All Rules could be kept in a single file if so desired. Bcfg2 simply uses the Groups in the files and priorities to determine how to assign Rules to a host's literal configuration.

```
<Rules priority="0">
    <Path type='directory' group="root" name="/autonfs" owner="root" perms="0755"/>
    <Path type='directory' group="utmp" name="/var/run/screen" owner="root" perms="0775"/>
```

```
    <Path type='directory' group="root" name="/autonfs/stage" owner="root" perms="0755"/>
    <Path type='directory' group="root" name="/exports" owner="root" perms="0755"/>
    <Path type='directory' name="/etc/condor" owner="root" group="root" perms="0755"/>
    <Path type='directory' name="/logs" group="wwwtrans" owner="root" perms="0775"/>
    <Path type='directory' name="/mnt" group="root" owner="root" perms="0755"/>
    <Path type='directory' name="/my" owner="root" group="root" perms="0755"/>
    <Path type='directory' name="/my/bin" owner="root" group="root" perms="0755"/>
    <Path type='directory' name="/nfs" owner="root" group="root" perms="0755"/>
    <Path type='directory' name="/sandbox" perms="0777" owner="root" group="root"/>
    <Path type='directory' name="/software" group="root" owner="root" perms="0755"/>
    <Path type='permissions' perms="0555" group="audio" owner="root" name="/dev/dsp"/>
    <Path type='permissions' perms="0555" group="audio" owner="root" name="/dev/mixer"/>
    <Path type='symlink' name="/bin/whatami" to="/mcs/adm/bin/whatami"/>
    <Path type='symlink' name="/chibahomes" to="/nfs/chiba-homefarm"/>
    <Path type='symlink' name="/home" to="/nfs/mcs-homefarm"/>
    <Path type='symlink' name="/homes" to="/home"/>
    <Path type='symlink' name="/mcs" to="/nfs/mcs"/>
    <Path type='symlink' name="/my/bin/bash" to="/bin/bash"/>
    <Path type='symlink' name="/my/bin/tcsh" to="/bin/tcsh"/>
    <Path type='symlink' name="/my/bin/zsh" to="/bin/zsh"/>
    <Path type='symlink' name="/software/common" to="/nfs/software-common"/>
    <Path type='symlink' name="/software/linux" to="/nfs/software-linux"/>
    <Path type='symlink' name="/software/linux-debian_sarge" to="/nfs/linux-debian_sarge"/>
    <Path type='symlink' name="/usr/bin/passwd" to="/usr/bin/yppasswd"/>
    <Path type='symlink' name="/usr/bin/yppasswd" to="/mcs/bin/passwd"/>
    <Path type='symlink' name="/usr/lib/libgd.so.1.8" to="/usr/lib/libgd.so.1.8.4"/>
    <Path type='symlink' name="/usr/lib/libtermcap.so.2" to="/usr/lib/libtermcap.so"/>
    <Path type='symlink' name="/usr/local/bin/perl" to="/usr/bin/perl"/>
    <Path type='symlink' name="/usr/local/bin/perl5" to="/usr/bin/perl"/>
    <Path type='symlink' name="/usr/local/bin/tcsh" to="/bin/tcsh"/>
    <Service name='ntpd' status='on' type='chkconfig'/>
    <Service name='haldaemon' status='on' type='chkconfig'/>
    <Service name='messagebus' status='on' type='chkconfig'/>
    <Service name='netfs' status='on' type='chkconfig'/>
    <Service name='network' status='on' type='chkconfig'/>
    <Service name='rawdevices' status='on' type='chkconfig'/>
    <Service name='sshd' status='on' type='chkconfig'/>
    <Service name='syslog' status='on' type='chkconfig'/>
    <Service name='vmware-tools' status='on' type='chkconfig'/>
</Rules>
```

SSHbase

SSHbase is a purpose-built Bcfg2 plugin for managing ssh host keys. It is responsible for making ssh keys persist beyond a client rebuild and building a proper `ssh_known_hosts file`, including a correct localhost record for the current system.

It has two functions:

- Generating new ssh keys – When a client requests a dsa, rsa, or v1 key, and there is no existing key in the repository, one is generated.

- Maintaining the `ssh_known_hosts` file – all current known public keys (and extra public key stores) are integrated into a single `ssh_known_hosts` file, and a localhost record for the current client is added. The `ssh_known_hosts` file data is updated whenever any keys change, are added, or deleted.

Interacting with SSHbase

- Pre-seeding with existing keys – Currently existing keys will be overwritten by new, sshbase-managed ones by default. Pre-existing keys can be added to the repository by putting them in <repo>/SSHbase/<key filename>.H_<hostname>
- Pre-seeding can also be performed using bcfg2-admin pull ConfigFile /name/of/ssh/key
- Revoking existing keys – deleting <repo>/SSHbase/*.H_<hostname> will remove keys for an existing client.

Aliases SSHbase has support for Aliases listed in clients.xml. The address for the entries are specified either through DNS (e.g. a CNAME), or via the address attribute to the Alias.

Getting started

1. Add SSHbase to the **plugins** line in `/etc/bcfg2.conf` and restart the server – This enables the SSHbase plugin on the Bcfg2 server.
2. Add Path entries for `/etc/ssh/ssh_known_hosts`, and `/etc/ssh/ssh_host_dsa_key`, etc to a bundle or base.
3. Enjoy.

At this point, SSHbase will generate new keys for any client without a recorded key in the repository, and will generate an `ssh_known_hosts` file appropriately.

Adding public keys for unmanaged hosts If you have some hosts which are not managed by Bcfg2, but you would still like to have their public ssh keys available in `ssh_known_hosts`, you can add their public keys to the `SSHbase` directory with a *.static* ending.

Example:

`a.static`:

```
TEST1
```

`b.static`:

```
TEST2
```

The generated `ssh_known_hosts` file:

```
TEST1
TEST2
```

Blog post http://www.ducea.com/2008/08/24/using-the-bcfg2-sshbase-plugin/

SSLCA

SSLCA is a generator plugin designed to handle creation of SSL private keys and certificates on request.

Borrowing ideas from the TGenshi and SSHbase plugins, SSLCA automates the generation of SSL certificates by allowing you to specify key and certificate definitions. Then, when a client requests a Path that contains such a definition within the SSLCA repository, the matching key/cert is generated, and stored in a hostfile in the repo so that subsequent requests do not result in repeated key/cert recreation. In the event that a new key or cert is needed, the offending hostfile can simply be removed from the repository, and the next time that host checks in, a new file will be created. If that file happens to be the key, any dependent certificates will also be regenerated.

Getting started In order to use SSLCA, you must first have at least one CA configured on your system. For details on setting up your own OpenSSL based CA, please see http://www.openssl.org/docs/apps/ca.html for details of the suggested directory layout and configuration directives.

For SSLCA to work, the openssl.cnf (or other configuration file) for that CA must contain full (not relative) paths.

1. Add SSLCA to the **plugins** line in `/etc/bcfg2.conf` and restart the server – This enabled the SSLCA plugin on the Bcfg2 server.

#. Add a section to your `/etc/bcfg2.conf` called sslca_foo, replacing foo with the name you wish to give your CA so you can reference it in certificate definitions.

#. Under that section, add an entry for `config` that gives the location of the openssl configuration file for your CA.

#. If necessary, add an entry for `passphrase` containing the passphrase for the CA's private key. We store this in `/etc/bcfg2.conf` as the permissions on that file should have it only readable by the bcfg2 user. If no passphrase is entry exists, it is assumed that the private key is stored unencrypted.

#. Add an entry `chaincert` that points to the location of your ssl chaining certificate. This is used when preexisting certifcate hostfiles are found, so that they can be validated and only regenerated if they no longer meet the specification.

#. Once all this is done, you should have a section in your `/etc/bcfg2.conf` that looks similar to the following:

> [sslca_default] config = /etc/pki/CA/openssl.cnf passphrase = youReallyThinkIdShareThis?
> chaincert = /etc/pki/CA/chaincert.crt

#. You are now ready to create key and certificate definitions. For this example we'll assume you've added Path entries for the key, `/etc/pki/tls/private/localhost.key`, and the certificate, `/etc/pki/tls/certs/localhost.crt` to a bundle or base.

#. Defining a key or certificate is similar to defining a TGenshi template. Under your Bcfg2's SSLCA directory, create the directory structure to match the path to your key. In this case this would be something like `/var/lib/bcfg2/SSLCA/etc/pki/tls/private/localhost.key`.

1. Within that directory, create a `key.xml` file containing the following:

```
<KeyInfo>
    <Key type="rsa" bits="2048" />
</KeyInfo>
```

#. This will cause the generation of an 2048 bit RSA key when a client requests that Path. Alternatively you can specify `dsa` as the keytype, or a different number of bits.

#. Similarly, create the matching directory structure for the certificate path, and a `cert.xml` containinng the following:

```
<CertInfo>
    <Cert format="pem" key="/etc/pki/tls/private/localhost.key" ca="default" days=
</CertInfo>
```

#. When a client requests the cert path, a certificate will be generated using the key hostfile at the specified key location, using the CA matching the ca attribute. ie. ca="default" will match [sslca_default] in your `/etc/bcfg2.conf`

TODO

1. Add generation of pkcs12 format certs

TCheetah

This document reflects the `TCheetah` plugin.

The `TCheetah` plugin allows you to use the cheetah templating system to create files, instead of the various diff-based methods offered by the `Cfg` plugin. It also allows you to include the results of probes executed on the client in the created files.

To begin, you will need to download and install the Cheetah templating engine from http://www.cheetahtemplate.org/. Once it is installed, you can enable it by adding `TCheetah` to the `plugins` line in `/etc/bcfg2.conf` on your Bcfg server. For example:

```
plugins = Cfg,Metadata,Pkgmgr,Rules,SSHbase,TCheetah
```

The `TCheetah` plugin makes use of a `Cfg`-like directory structure located in in a `TCheetah` subdirectory of your repository, usually `/var/lib/bcfg2/TCheetah`. Each file has a directory containing two files, `template` and `info`. The template is a standard Cheetah template with two additions:

- *self.metadata* is the client's metadata
- *self.properties* is an xml document of unstructured data

The `info` file is formatted like `:info` files from Cfg.

Mostly, people will want to use client metadata.

File permissions File permissions for entries handled by TCheetah are controlled via the use of *Info* files. Note that you **cannot** use both a Permissions entry and a Path entry to handle the same file.

self.metadata variables The following variables are available for self.metadata:

- hostname
- bundles
- groups
- categories
- probes
- uuid
- password

self.metadata is an instance of the class ClientMetadata of file Bcfg2/Server/Plugins/Metadata.py.

self.properties Properties is a python ElementTree object, loaded from the data in `/var/lib/bcfg2/Properties/<properties file>.xml`. That file should have a `Properties` node at its root.

Example `Properties/example.xml`:

```
<Properties>
 <host>
   <www.example.com>
     <rootdev>/dev/sda</rootdev>
   </www.example.com>
 </host>
</Properties>
```

You may use any of the ElementTree methods to access data in your template. Several examples follow, each producing an identical result on the host 'www.example.com':

```
$self.Properties['example.xml'].find('host').find('www.example.com').find('rootdev').text
$self.Properties['example.xml'].find('host').find($self.metadata.hostname).find('rootdev')
${self.Properties['example.xml'].xpath('host/www.example.com/rootdev')[0].text}
${self.Properties['example.xml'].xpath('host/' + self.metadata.hostname + '/roctdev')[0].te
#set $path = 'host/' + $self.metadata.hostname + '/rootdev'
${self.Properties['example.xml'].xpath($path)[0].text}
${self.Properties['example.xml'].xpath(path)[0].text}
```

Simple Example TCheetah works similar to Cfg in that you define all literal information about a particular file in a directory rooted at TGenshi/path_to_file. The actual file contents are placed in a file named *template* in that directory. Below is a simple example a file `/foo`.

`/var/lib/bcfg2/TCheetah/foo/template`

```
> buildfile /foo <clientname>
Hostname is $self.metadata.hostname
Groups:
#for $group in $self.metadata.groups:
 * $group
```

```
#end for
Categories:
#for $category in $self.metadata.categories:
 * $category -- $self.metadata.categories[$category]
#end for

Probes:
#for $probe in $self.metadata.Probes:
 * $probe -- $self.metadata.Probes[$probe]
#end for

/var/lib/bcfg2/TCheetah/foo/info

perms: 624
```

Output The following output can be generated with bcfg2-info. Note that probe information is not persistent, hence, it only works when clients directly query the server. For this reason, bcfg2-info output doesn't reflect current client probe state.

```
<Path type="file" name="/foo" owner="root" perms="0624" group="root">
Hostname is topaz.mcs.anl.gov
Groups:
 * desktop
 * mcs-base
 * ypbound
 * workstation
 * xserver
 * debian-sarge
 * debian
 * a
Categories:
 * test -- a

Probes:
</Path>
```

Example: Replace the crontab plugin In many cases you can use the TCheetah plugin to avoid writing custom plugins in Python. This example randomizes the time of cron.daily execution with a stable result. Cron.daily is run at a consistent, randomized time between midnight and 7am.:

```
#import random
#silent random.seed($self.metadata.hostname)

# /etc/crontab: system-wide crontab
# Unlike any other crontab you don't have to run the `crontab`
# command to install the new version when you edit this file.
# This file also has a username field, that none of the other crontabs do.

SHELL=/bin/sh
PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin://bin
```

```
# m h dom mon dow user  command
17 *    * * *   root    run-parts --report /etc/cron.hourly
$random.randrange(0,59) $random.randrange(0,6)    * * *   root    test -x /usr/sbin/anacron
47 6    * * 7   root    test -x /usr/sbin/anacron || run-parts --report /etc/cron.weekly
52 6    1 * *   root    test -x /usr/sbin/anacron || run-parts --report /etc/cron.monthly.
```

Note: Comments and Cheetah As Cheetah processes your templates it will consider hash "#" style comments to be actual comments in the template and will strip them from the final config file. If you would like to preserve the comment in the final config file you need to escape the hash character '#' which will tell Cheetah (and Python) that you do in fact want the comment to appear in the final config file.:

```
# This is a comment in my template which will be stripped when it's processed through Cheet
\# This comment will appear in the generated config file.
```

Each of these plugins has a corresponding subdirectory with the same name in the Bcfg2 repository.

Statistics Plugins

DBStats

DBStats can be enabled by adding DBStats to the plugins line in `/etc/bcfg2.conf`:

plugins = DBStats

For more information on how to use DBStats to setup reporting, see *Bcfg2 Dynamic Reporting System*.

Statistics

DBStats can be enabled by adding it to the plugins line in */etc/bcfg2.conf*.

Version Plugins

Bzr

Why use the Bazaar plugin The Bazaar plugin is useful if you would like to track changes to your bcfg2 repository using a Bazaar backend. Currently, it enables you to get revision information out of your repository for reporting purposes. Future plans are to commit changes to the repo which are made by the server.

How to enable the Bazaar plugin Simply add "Bzr" to your plugins line in `/etc/bcfg2.conf`:

```
[server]
plugins = Base,Bundler,Cfg,...,Bzr
```

Usage notes Unlike other VCS plugins for Bcfg2, the Bazaar plugin checks whether there are uncommitted changes to the repository. If there are, this plugin appends a "+" after the version number. Essentially, this means you're using that version, "plus" some changes.

CVS

Why use the CVS plugin The CVS plugin is useful if you would like to track changes to your Bcfg2 repository using a CVS backend. Currently, it enables you to get revision information out of your repository for reporting purposes. Future plans are to commit changes to the repo which are made by the server.

How to enable the CVS plugin Simply add "Cvs" to your plugins line in `/etc/bcfg2.conf`:

```
[server]
plugins = Base,Bundler,Cfg,...,Cvs
```

Darcs

This page describes the new Darcs plugin which is experimental.

Why use the Darcs plugin The Darcs plugin is useful if you would like to track changes to your Bcfg2 repository using a Darcs backend. Currently, it enables you to get revision information out of your repository for reporting purposes. Once the plugin is enabled, every time a client checks in, it will include the current repository revision in the reports/statistics.

How to enable the Darcs plugin You will need to install Darcs on the Bcfg2 server first. Once installed, simply add Darcs to your plugins line in `/etc/bcfg2.conf`:

```
[server]
plugins = Base,Bundler,Cfg,...,Darcs
```

Fossil

Why use the Fossil plugin The Fossil plugin is useful if you would like to track changes to your bcfg2 repository using a Fossil SCM backend. Currently, It enables you to get revision information out of your repository for reporting purposes. Future plans are to commit changes to the repo which are made by the server.

How to enable the Fossil plugin Simply add "Fossil" to your plugins line in `/etc/bcfg2.conf`:

```
[server]
plugins = Base,Bundler,Cfg,...,Fossil
```

Git

Why use the Git plugin The Git plugin is useful if you would like to track changes to your bcfg2 repository using a Git backend. Currently, It enables you to get revision information out of your repository for reporting purposes. Once the plugin is enabled, every time a client checks in, it will include the current repository revision in the reports/statistics.

Future plans are to commit changes to the repo which are made by the server (adding clients, ssh keys, etc).

How to enable the Git plugin The Git plugin uses Dulwich to interface with git repositories. Therefore, you will need to install Dulwich on the Bcfg2 server first. Once installed, simply add Git to your plugins line in `/etc/bcfg2.conf`:

```
[server]
plugins = Base,Bundler,Cfg,...,Git
```

Mercurial (Hg)

Why use the Mercurial plugin The Hg plugin is useful if you would like to track changes to your Bcfg2 repository using Hg backend. Currently, it enables you to get revision information out of your repository for reporting purposes.

How to enable the Mercurial plugin You will need to install Mercurial on the Bcfg2 server first.

Simply add Hg to your plugins line in `/etc/bcfg2.conf`:

```
[server]
plugins = Base,Bundler,Cfg,...,Hg
```

Svn

The Svn plugin is useful if you would like to track changes to your bcfg2 repository using a Subversion backend. It deprecates the previous Subversion integration mentioned here at ftp://ftp.mcs.anl.gov/pub/bcfg/papers/directing-change-with-bcfg2.pdf.

Currently, It enables you to get revision information out of your repository for reporting purposes. Once the plugin is enabled, every time a client checks in, it will include the current repository revision in the reports/statistics.

Future plans are to commit changes to the repo which are made by the server (adding clients, ssh keys, etc).

How to enable the Svn plugin Simply add Svn to your plugins line in `/etc/bcfg2.conf`:

```
[server]
plugins = Base,Bundler,Cfg,..,Svn
```

5.1.3 Plugin Roles (in 1.0)

In version 1.0, plugins have been refactored into a series of roles. This are fine-grained plugin capabilities that govern how the server core interacts with plugins.

More details can be found in *Plugin Roles*

Plugin Roles

This documents available plugin roles.

1. list of plugin roles

Role	Class	Status
Metadata	Metadata	done
Connector	Connector	done
Probing	Probing	done
Structure	Structure	done
Structure Val	StructureValidator	done
Generator	Generator	done
Goals Val	GoalValidator	done
Statistics	Statistics	done
Pull Source	PullSource	done
Pull Target	PullTarget	done
Version	Version	done
Decision	Decision	done
Remote	Remote	none
Syncing	Syncing	none

2. Plugin Capabilities
   - Metadata
     - Initial metadata construction
     - Connector data accumulation
     - ClientMetadata instance delivery
     - Introspection interface (for bcfg2-info & co)
   - Connector
     - Provide additional data for ClientMetadata instances
   - Probing
     - send executable probes to clients and receive data responses
   - Structure
     - Produce a list of configuration entries that should be included in client configurations
     - Each structure plugin is produces a list of structures

    - Core verifies that each bundle listed has been constructed
  - Structure Validation
    - Validate a client entry list's internal consistency, modifying if needed
  - Generator
  - Goals Validation
    - Validate client goals, modifying if needed
  - Pull Source
    - Plugin can provide entry information about clients
  - Pull Target
    - Plugin can accept entry data and merge it into the specification
  - Version
    - Plugin can read revision information from VCS of choice
    - Will provide an interface for producing commits made by the bcfg2-server
  - Decision

3. Configuration of plugins

   Plugin configuration will be simplified substantially. Now, a single list of plugins (including plugins of all capabilities) is specified upon startup (either via bcfg2.conf or equivalent). This mechanism replaces the current split configuration mechanism where generators, structures, and other plugins are listed independently. Instead, all plugins included in the startup list will be initialized, and each will be enabled in all roles that it supports. This will remove a current source of confusion and potential configuration errors, wherein a plugin is enabled for an improper set of goals. (ie Cfg enabled as a structure, etc) This does remove the possibility of partially enabling a plugin for one of its roles without activating it across the board, but I think this is a corner case, which will be poorly supported by plugin implementers. If needed, this use case can be explicitly supported by the plugin author, through use of a config file directive.

4. User Visible Changes

   Connector data is added to ClientMetadata instances using the name of the connector plugin. This means that the dictionary of key/val probe pairs included with metadata is now available as metadata.Probes (instead of metadata.probes). Once properties are available the same way, they will likewise change names to metadata.Properties from their current name.

   Plugin configuration will change. A single field "plugins" in bcfg2.conf will supercede the combination of the "generators" and "structures" fields.

   Default loading of needed plugins is now explicit; this means that Statistics (if used) should be listed in the plugins line of bcfg2.conf.

5. Notes

   - Need to ensure bundle accumulation occurs with connector groups

Probes

At times you need to gather information from a client machine before you can generate its configuration. For example, if some of your machines have both a local scratch disk and a system disk while others only have the system disk, you would want to know this information to correctly generate an */etc/auto.master* autofs config file for each type. Here we will look at how to do this.

First you will need to set up the TCheetah plugin, as described on the *TCheetah* page.

Next, we need to create a `Probes` directory in our toplevel repository location:

```
mkdir /var/lib/bcfg2/Probes
```

This directory will hold any small scripts we want to use to grab information from client machines. These scripts can be in any scripting language; the shebang line (the `#!/usr/bin/env some_interpreter_binary` line at the very top of the script) is used to determine the script's interpreter.

Note: Bcfg2 uses python mkstemp to create the Probe scripts on the client. If your /tmp directory is mounted **noexec**, you will likely need to modify the TMPDIR environment variable so that the bcfg2 client creates the temporary files in a directory from which it can execute.

Now we need to figure out what exactly we want to do. In this case, we want to hand out an `/etc/auto.master` file that looks like:

```
/software  /etc/auto.software --timeout 3600
/home      /etc/auto.home --timeout 3600
/hometest  /etc/auto.hometest --timeout 3600
/nfs       /etc/auto.nfs --timeout 3600
/scratch   /etc/auto.scratch --timeout 3600
```

for machines that have a scratch disk. For machines without an extra disk, we want to get rid of that last line:

```
/software  /etc/auto.software --timeout 3600
/home      /etc/auto.home --timeout 3600
/hometest  /etc/auto.hometest --timeout 3600
/nfs       /etc/auto.nfs --timeout 3600
```

So, from the Probes standpoint we want to create a script that counts the number of SCSI disks in a client machine. To do this, we create a very simple `Probes/scratchlocal` script:

```
cat /proc/scsi/scsi | grep Vendor | wc -l
```

Running this on a node with *n* disks will return the number *n+1*, as it also counts the controller as a device. To differentiate between the two classes of machines we care about, we just need to check the output of this script for numbers greater than 2. We do this in the template.

The `TCheetah/` directory is laid out much like the `Cfg/` directory. For this example we will want to create a `TCheetah/etc/auto.master` directory to hold the template of the file in question. Inside of this template we will need to check the result of the Probe script that got run and act accordingly. The `TCheetah/etc/auto.master/template` file looks like:

```
/software  /etc/auto.software --timeout 3600
/home      /etc/auto.home --timeout 3600
/hometest  /etc/auto.hometest --timeout 3600
/nfs       /etc/auto.nfs --timeout 3600
#if int($self.metadata.Probes["scratchlocal"]) > 2
/scratch   /etc/auto.scratch --timeout 3600
#end if
```

Any Probe script you run will store its output in `$self.metadata.Probes["scriptname"]`, so we get to our *scratchlocal* script's output as seen above. Note that we had to wrap the output in an *int()* call; the script output is treated as a string, so it needs to be converted before it can be tested numerically.

With all of these pieces in place, the following series of events will happen when the client is run:

1. Client runs
2. Server hands down our `scratchlocal` probe script
3. Client runs the `scratchlocal` probe script and hands its output back up to the server
4. Server generates `/etc/auto.master` from its template, performing any templating substitutions/actions needed in the process.
5. Server hands `/etc/auto.master` down to the client
6. Client puts file contents in place.

Now we have a nicely dynamic `/etc/auto.master` that can gracefully handle machines with different numbers of disks. All that's left to do is to add the `/etc/auto.master` to a Bundle:

```
<Path name='/etc/auto.master'/>
```

Host and Group Specific probes

Bcfg2 has the ability to alter probes based on client hostname and group membership. These files work similarly to files in Cfg.

If multiple files with the same basename apply to a client, the most specific one is used. Only one instance of a probe is served to a given client, so if a host-specific version and generic version apply, only the client-specific one will be used.

Other examples

current-kernel Probe the currently running kernel.

```
#!/bin/sh
#
# PROBE_NAME : current-kernel
echo `uname -r`
```

group Probe used to dynamically set client groups based on OS/distro.

Note: Some parts of this script may depend on having lsb-release installed.

```
#!/bin/bash

OUTPUT=""

if [ -e /etc/release ]; then
        # Solaris
        OUTPUT=$OUTPUT'\n'`echo group:solaris`
elif [ -e /etc/debian_version ]; then
        # debian based
        OUTPUT=$OUTPUT'\n'`echo group:deb`
        if [ -e /etc/lsb-release ]; then
                # variant
                . /etc/lsb-release
                OS_GROUP=$DISTRIB_CODENAME
                DEBIAN_VERSION=$(echo "$DISTRIB_ID" | tr '[A-Z]' '[a-z]')
                case "$OS_GROUP" in
                        "lucid")
                        OUTPUT=$OUTPUT'\n'`echo group:$DISTRIB_CODENAME`
                        OUTPUT=$OUTPUT'\n'`echo group:$DEBIAN_VERSION`
                        ;;
                esac
        else
                # debian
                OS_GROUP=`cat /etc/debian_version`
                OUTPUT=$OUTPUT'\n'`echo group:debian`
                case "$OS_GROUP" in
                        5.*)
                        OUTPUT=$OUTPUT'\n'`echo group:lenny`
                        ;;
                        "sid")
                        OUTPUT=$OUTPUT'\n'`echo group:sid`
                        ;;
                esac
        fi
elif [ -e /etc/redhat-release ]; then
        # redhat based
        OUTPUT=$OUTPUT'\n'`echo group:rpm`
        OS_GROUP=`cat /etc/redhat-release | cut -d' ' -f1 | tr '[A-Z]' '[a-z]'`
        REDHAT_VERSION=`cat /etc/redhat-release | cut -d' ' -f3`
        case "$OS_GROUP" in
                "centos" | "fedora")
                OUTPUT=$OUTPUT'\n'`echo group:$OS_GROUP`
                OUTPUT=$OUTPUT'\n'`echo group:$OS_GROUP$REDHAT_VERSION`
                ;;
        esac
elif [ -e /etc/gentoo-release ]; then
        # gentoo
        OUTPUT=$OUTPUT'\n'`echo group:gentoo`
elif [ -x /usr/sbin/system_profiler ]; then
        # os x
```

```
        ### NOTE: Think about using system_profiler SPSoftwareDataType here
        OUTPUT=$OUTPUT'\n'`echo group:osx`
        OSX_VERSION=`sw_vers | grep 'ProductVersion:' | egrep -o '[0-9]+\.[0-9]+'`
        if [ "$OSX_VERSION" == "10.6" ]; then
                OUTPUT=$OUTPUT'\n'`echo group:osx-snow`
        elif [ "$OSX_VERSION" == "10.5" ]; then
                OUTPUT=$OUTPUT'\n'`echo group:osx-leo`
        fi
        echo $OUTPUT
else
        exit 0
fi
# get the proper architecture
ARCH=`uname -m`
case "$ARCH" in
        "x86_64")
        if [ "$OS_GROUP" == 'centos' ]; then
                OUTPUT=$OUTPUT'\n'`echo group:$ARCH`
        else
                OUTPUT=$OUTPUT'\n'`echo group:amd64`
        fi
        ;;
        "i386" | "i686")
        OUTPUT=$OUTPUT'\n'`echo group:i386`
        ;;
        "sparc64")
        OUTPUT=$OUTPUT'\n'`echo group:sparc64`
        ;;
esac

# output the result of all the group probing
# (interpreting the backslashed newlines)
echo -e $OUTPUT
```

vserver Detect if the server is a Linux-VServer host.

```
#!/bin/sh

# Test the proc
TEST=`cat /proc/self/status|grep s_context| cut -d":" -f2|cut -d" " -f 2`

case "$TEST" in
  "")
    # Not a vserver kernel
    echo group:host
    ;;
  "0")
    # Vserver kernel but it is the HOST
    echo group:host
    ;;
  [0-9]*)
    # Vserver
```

```
    echo group:vserver
    ;;
esac
```

grub-serial-order A basic hardware probe to determine if you should change the default serial ordering in grub.conf. This pre-supposes that you know your hardware is broken. You can tell something is wrong with your hardware if it takes lots of time to iterate through the "Press a key" option and present you with the grub menu. In some cases, I've seen this take as long as 20 minutes.

```
#!/bin/sh
#
#
# We need to modify the order of the --serial line in grub
# in order to fix silly hardware bugs.  In some cases, having
# this in the wrong order causes grub to take an inordinate
# amount of time to do anything before it actually auto-picks
# the default menu option to boot.
#

PATH=/bin:/usr/bin:/sbin:/usr/sbin; export PATH
# let's figure out what product type this is
os=`uname -s`
productname="product-no-dmidecode"

if [ $os = "Linux" ] ; then
    productname=`dmidecode -s system-product-name 2>&1`
    case $productname in
    "PowerEdge M600")
        echo "console serial"
        ;;
    *)
        echo "serial console"
        ;;
    esac
fi
if [ $os = "SunOS" ] ; then
   # Bcfg2 server is unhappy with null output from probes
   echo "console"
fi
```

manufacturer Probe to output some standardized group names based on the manufacturer information.

```
#!/bin/sh
#
PATH=/bin:/usr/bin:/sbin:/usr/sbin; export PATH

manufacturer=manuf-no-demidecode

os=`uname -s`
if [ $os = "Linux" ] ; then
    manufacturer=`dmidecode -s system-manufacturer 2>&1| sed -e 's/[ ]\+$//g'`
```

```
    case $manufacturer in
    "Dell Inc.")
        manufacturer="manuf-dell"
        ;;
    "Sun Microsystems")
        manufacturer="manuf-sun"
        ;;
    "VMware, Inc.")
        manufacturer="manuf-vmware"
        ;;
    *)
        manufacturer="manuf-unknown"
        ;;
    esac
fi

if [ $os = "SunOS" ]; then
    case `uname -i` in
    SUNW,*)
        manufacturer="manuf-sun"
        ;;
    *)
        manufacturer="manuf-unknown"
        ;;
    esac
fi

echo group:$manufacturer
```

producttype A probe to set up dynamic groups based on the producttype and possibly some internal components of the system.

Defined products are product-name.

Defined component information is has_some_component. In the example below, we can infer that we have Emulex Lightpulse gear and set the group has_hardware_emulex_lightpulse.

```
!/bin/sh
#
#

PATH=/bin:/usr/bin:/sbin:/usr/sbin; export PATH
# let's figure out what product type this is
os=`uname -s`
productname="product-no-dmidecode"

if [ $os = "Linux" ] ; then
    productname=`dmidecode -s system-product-name 2>&1`
    case $productname in
    "PowerEdge M600")
        productname="product-blademe600"
        ;;
    "Sun Fire X4100 M2")
```

```
        productname="product-x4100m2"
        ;;
    "Sun Fire X4440")
        productname="product-x4440"
        ;;
    "VMware Virtual Platform")
        productname="product-vmware-vm"
        ;;
    *)
        productname="product-unknown"
        ;;
    esac

    # check for emulex lightpulse fiber channel HBA
    check_emulex_lightpulse=`lspci -d 10df: | grep -c LightPulse`
    if [ $check_emulex_lightpulse -gt 0 ]; then
        echo group:has_hardware_emulex_lightpulse
    fi

    # check for broadcom nics
    check_broadcom_nic=`lspci -d 14e4: | grep -c NetXtreme`
    if [ $check_broadcom_nic -gt 0 ]; then
        echo group:has_hardware_broadcom_nic
    fi

    # check for intel pro/1000 MT nics
    check_intel_pro1000mt_nic=`lspci -d 8086:1010 | wc -l`
    if [ $check_intel_pro1000mt_nic -gt 0 ]; then
        echo group:has_hardware_intel_pro1000mt_nic
    fi

fi

if [ $os = "SunOS" ] ; then
    case `uname -i` in
    SUNW,*)
        productname=`uname -i`
        ;;
    *)
        productname=product-unknown
        ;;
    esac
fi

echo group:$productname
```

serial-console-speed A probe to tell us what the serial console speed should be for a given piece of hardware. This pre-supposed some knowledge of the hardware because you define the speeds in here instead of attempting to probe bios or something in the hardware in most cases (like x86).

```
#!/bin/sh
#
```

```
#
# figure out what serial speed we should tell bcfg2 to use.
# since there's no way to probe, we need to set this up by external
# knowledge of the system hardware type (and just make sure we
# standardize on that serial speed for that hardware class)

PATH=/bin:/usr/bin:/sbin:/usr/sbin; export PATH
# let's figure out what product type this is
os=`uname -s`
productname="product-no-dmidecode"

if [ $os = "Linux" ] ; then
    productname=`dmidecode -s system-product-name 2>&1`
    case $productname in
    "PowerEdge M600")
        echo "115200"
        ;;
    *)
        echo "9600"
        ;;
    esac
fi
if [ $os = "SunOS" ]; then
    platform=`uname -i`
    case $platform in
    SUNW,*)
    eeprom ttya-mode | sed 's/ttya-mode=//'|awk -F, '{print $1}'
    ;;
    *)
    echo "9600"
    ;;
    esac
fi
```

Ohai probes

The Ohai plugin is used to detect information about the client operating system. The data is reported back to the server using JSON.

Client prerequisites

On the client, you need to install Ohai. See Ohai-Install for more information.

Server prerequisites

If you have python 2.6 or later installed, you can continue on to *Setup*. Otherwise, you will need to install the python-simplejson module found packaged in most distributions.

Setup

To enable the Ohai plugin, you need to first create an `Ohai` directory in your Bcfg2 repository (e.g. `/var/lib/bcfg2/Ohai`). You then need to add **Ohai** to the plugins line in `bcfg2.conf`. Once this is done, restart the server and start a client run. You will have the JSON output from the client in the `Ohai` directory you created previously.

Trigger

Trigger is a plugin that calls external scripts (on the server) when clients are configured.

Setup

First, add Trigger to the **plugins** line in bcfg2.conf. Then do the following:

```
mkdir /var/lib/bcfg2/Trigger
echo "#!/bin/sh\necho $1\n" > /var/lib/bcfg2/Trigger/test.sh
chmod +x /var/lib/bcfg2/Trigger/test.sh
```

Use cases

1. Completing network builds (ie resetting from the build target to the boot pxe target)
2. Integration with external systems

Trigger Arguments

Triggers are run with a series of arguments.

1. client hostname
2. -p
3. client profile
4. -g
5. group1:group2:...:groupN (all client groups)

5.2 Admin

The `bcfg2-admin` command provides you an interface which allows you to interact with your Bcfg2 repository in an administrative fashion. To get started, run `bcfg2-admin help`. You will be presented with a list of different *modes* which each provide various administrative functionality. Available modes are listed below.

FIXME: Need examples for each command listed below.

5.2.1 client

Create, delete, or modify client entries.

5.2.2 compare

Determine differences between files or directories of client specification instances.

5.2.3 init

Interactively initialize a new repository.

5.2.4 minestruct

Extract extra entry lists from statistics.

5.2.5 perf

Query server for performance data.

5.2.6 pull

Integrate configuration information from clients into the server repository.

5.2.7 query

Query clients.

The default result format is suitable for consumption by pdsh. This example queries the server for all clients in the *ubuntu* group:

```
bcfg2-admin query g=ubuntu
```

5.2.8 snapshots

Interact with the Snapshots system.

5.2.9 tidy

Clean up useless files in the repo.

5.2.10 viz

Produce graphviz diagrams of metadata structures.

The following command will produce a graphviz image which includes hosts, bundles, and a key:

```
bcfg2-admin viz -H -b -k -o ~/bcfg2.png
```

Note: The graphviz package available via DAG/RPMforge has been known to have dependency issues. We recommend installing the package from EPEL.

5.2.11 xcmd

XML-RPC Command Interface.

5.3 Configuration Entries

This page describes the names and semantics of each of the configuration entries used by Bcfg2.

5.3.1 Non-POSIX entries

TagName	Description	Attributes
Action	Command	name, command, when, timing
Package	Software Packages	name, type, version, url
PostInstall	PostInstall command	name
Service	System Services	name, type, status, reload

5.3.2 POSIX entries

New in version 1.0.0. The unified POSIX Path entries prevent inconsistent configuration specifications of multiple entries for a given path. The following table describes the various types available for new **Path** entries.

The abstract specification of these entries (i.e. In Bundler) will only contain a *name* attribute. The type will be added by the plugin that handles the entry in the case of Cfg, TGenshi, or TCheetah. If the entry is handled by the Rules plugin (i.e. it is a device, directory, hardlink, symlink, etc), then you will specify both the *type* and any other necessary attributes in Rules.

Running `bcfg2-repo-validate` will check your configuration specification for the presence of any mandatory attributes that are necessary for the Path type specified.

Note: A tool for converting old POSIX entries is available in the Bcfg2 source directory at tools/posixunified.py

Type	Replacement/New	Description	Attributes
device	New	Create block, character, and fifo devices	name, owner, group, dev_type (block, char, fifo), major/minor (for block/char devices)
directory	Replaces Directory entries	Directories	name, owner, group, perms, prune
file	Replaces ConfigFile entries	Configuration File	name, owner, group, perms, encoding, empty
hardlink	New	Create hardlinks	name, to
symlink	Replaces SymLink entries	SymLinks	name, to
ignore	New	Ignore files that cause package verification failures (currently applies to only YUMng)	name
nonexistent	New	Specify a path that should not exist	name
permissions	Replaces Permissions entries	Permissions of POSIX entities	name, owner, group, perms

Keep in mind that permissions for files served up by Cfg/TGenshi/TCheetah are still handled via the traditional *Info* mechanisms.

5.3.3 Bound Entries

This feature is a mechanism to specify a full entry at once from a bundle. Traditionally, entries are defined in two stages. First, an abstract entry is defined in a bundle. This entry includes a type (the XML tag) and a name attribute. Then this entry is bound for a client, providing the appropriate instance of that entry for the client. Specifying a bound entry short-circuits this process; the only second stage processing on Bound entries is to remove the "Bound" prefix from the element tag. The use of a bound entry allows the single stage definition of a complete entry. Bound entries can be used for any type.

Example:

```
<Bundle name='ntp'>
   <BoundPackage name='ntp' type='deb' version='1:4.2.4p4+dfsg-3ubuntu2.1'/>
</Bundle>
```

5.3.4 Fun and Profit using altsrc

Altsrc is a generic, bcfg2-server-side mechanism for performing configuration entry name remapping for the purpose of data binding.

Use Cases

- Equivalent configuration entries on different architectures with different names
- Mapping entries with the same name to different bind results in a configuration (two packages with the same name but different types)
- A single configuration entry across multiple specifications (multi-plugin, or multi-repo)

Examples

- Consider the case of `/etc/hosts` on linux and `/etc/inet/hosts` on solaris. These files contain the same data in the same format, and should typically be synchronized, however, exist in different locations. Classically, one would need to create one entry for each in Cfg or TCheetah and perform manual synchronization. Or, you could use symlinks and pray. Altsrc is driven from the bundle side. For example:

```
<Bundle name='netinfo'>
   <Group name='solaris'>
     <Path name='/etc/inet/hosts' altsrc='/etc/hosts'/>
   </Group>
   <Group name='linux'>
     <Path name='/etc/hosts'/>
   </Group>
</Bundle>
```

  In this case, when a solaris host gets the 'netinfo' bundle, it will get the first Path entry, which includes an altsrc parameter. This will cause the server to bind the entry as if it were a Path called `/etc/hosts`. This configuration entry is still called `/etc/inet/hosts`, and is installed as such.

- On encap systems, frequently multiple packages of the same name, but of different types will exist. For example, there might be an openssl encap package, and an openssl rpm package. This can be dealt with using a bundle like:

```
<Bundle name='openssl'>
   <Package name='openssl' altsrc='openssl-encap'/>
   <Package name='openssl' altsrc='openssl-rpm'/>
</Bundle>
```

  This bundle will bind data for the packages "openssl-encap" and "openssl-rpm", but will be delivered to the client with both packages named "openssl" with different types.

- Finally, consider the case where there exist complicated, but completely independent specifications for the same configuration entry but different groups of clients. The following bundle will allow the use of two different TCheetah templates `/etc/firewall-rules-external` and `/etc/firewall-rules-internal` for different clients based on their group membership.

```
<Bundle name='firewall'>
   ...
   <Group name='conduit'>
     <Path name='/etc/firewall-rules' altsrc='/etc/firewall-rules-external'/>
   </Group>
```

```
    <Group name='internal'>
      <Path name='/etc/firewall-rules' altsrc='/etc/firewall-rules-internal'/>
    </Group>
</Bundle>
```

- Consider the case where a variety of files can be constructed by a single template (TCheetah or TGenshi). It would be possible to copy this template into the proper location for each file, but that requires proper synchronization upon modification and knowing up front what the files will all be called. Instead, the following bundle allows the use of a single template for all proper config file instances.

  ```
  <Bundle name='netconfig'>
    <Path name='/etc/sysconfig/network-scripts/ifcfg-eth0' altsrc='/etc/ifcfg-template
    <Path name='/etc/sysconfig/network-scripts/ifcfg-eth1' altsrc='/etc/ifcfg-template
    <Path name='/etc/sysconfig/network-scripts/ifcfg-eth2' altsrc='/etc/ifcfg-template
  </Bundle>
  ```

  altsrc can be used as a parameter for any entry type, and can be used in any structure, including Bundler and Base.

5.4 Info

Various file properties for entries served by the Cfg, TGenshi, and TCheetah plugins are controlled through the use of `:info`, `info`, or `info.xml` files.

By default, these plugins are set to write files to the filesystem with owner **root**, group **root**, and mode **644** (read and write for owner, read only for group and other). These options, and a few others, can be overridden through use of `:info` or `info` files. Each config file directory can have a `:info` or `info` file if needed. The possible fields in an info file are:

Field	Possible values	Description	Default
encoding:	ascii | base64	Encoding of the file. Use base64 for non-ASCII files	ascii
group:	Any valid group	Sets group of the file	root
important:	true | false	Important entries are installed first during client execution	root
owner:	Any valid user	Sets owner of the file	root
paranoid:	yes | no	Backup file before replacement?	no
perms:	Numeric file mode	Sets the permissions of the file	0644

A sample info file for CGI script on a web server might look like:

```
owner: www
group: www
perms: 0755
```

Back to the fstab example again, our final `Cfg/etc/fstab/` directory might look like:

```
:info
fstab
fstab.G50_server
fstab.G99_fileserver
fstab.H_host.example.com
```

5.4.1 Important attribute

New in version 1.1.0. Having important entries hardcoded into the various client tools has worked relatively well so far. However, this method allows for a bit more flexibility as the entries can be controlled via the configuration specification.

Field	Possible values	Description	Default
important:	true | false	Important entries are installed first during client execution	root

5.4.2 info.xml files

`info.xml` files add the ability to specify different sets of file metadata on a group by group basis. These files are XML, and work similarly to those used by *Rules* or *Pkgmgr*.

The following specifies a different global set of permissions (root/sys/0651) than on clients in group webserver (root/root/0652)

```
<FileInfo>
  <Group name='webserver'>
    <Info owner='root' group='root' perms='0652'/>
  </Group>
  <Info owner='root' group='sys' perms='0651'/>
</FileInfo>
```

5.5 Bcfg2 Snapshots

New in version 1.0.0. This page describes the Snapshots plugin. This plugin is meant to replace the older *Bcfg2 Dynamic Reporting System*. It stores various aspects of a client's state when the client checks into the server.

5.5.1 Before you begin

Make sure you have version 0.5 or greater of sqlalchemy.

On CentOS/RHEL 5

- Download a tarball of SQLAlchemy.
- Extract and build the RPM:

```
tar xzf SQLAlchemy-0.5.6.tar.gz
cd SQLAlchemy-0.5.6
python setup.py bdist_rpm
```

- Copy the RPM in `SQLAlchemy-0.5.6/dist/` to your Yum repository, and rebuild the repository using `createrepo`.
- Clear the Yum cache:

```
sudo yum clean all
```

- Install SQLAlchemy:

```
sudo yum install SQLAlchemy
```

- Manage the package in Bcfg2 as you would any other package.

5.5.2 Configuration

- A database location needs to be added to `bcfg2.conf`. Three drivers are currently supported; mysql, postgres, and sqlite. When using the sqlite driver, only the driver and database lines are required.
  - For MySQL:

```
[snapshots]
driver = mysql
database = snapshots
user = snapshots
password = snapshots
host = dbserver
```

  - For SQLite:

```
[snapshots]
driver = sqlite
database = /var/lib/bcfg2/var/snapshots.sqlite
```

- The database needs to be initialized.:

```
$ bcfg2-admin snapshots init
2009-03-22 21:40:24,683 INFO sqlalchemy.engine.base.Engine.0x...3e2c PRAGMA table_ir
PRAGMA table_info("connkeyval")
2009-03-22 21:40:24,684 INFO sqlalchemy.engine.base.Engine.0x...3e2c ()
()
2009-03-22 21:40:24,686 INFO sqlalchemy.engine.base.Engine.0x...3e2c PRAGMA table_ir
PRAGMA table_info("package")
2009-03-22 21:40:24,687 INFO sqlalchemy.engine.base.Engine.0x...3e2c ()
()
.....
COMMIT
```

- The Snapshots plugin needs to be enabled for the bcfg2-server (by adding Snapshots to the plugins line in `/etc/bcfg2.conf`). Once done, this will cause the the server to store statistics information when clients run.

5.5.3 Using the reports interface

All hosts:

```
$ bcfg2-admin snapshots reports -a

============ ========= ========================================== =======================
Client       Correct   Revision                                   Time
============ ========= ========================================== =======================
bcfg2client  True      f46ac7773712bd3c3cfb765ae5d2a3b2a37ac9b7   2009-04-23 11:27:54.378
============ ========= ========================================== =======================
```

List bad entries for a single host:

```
$ bcfg2-admin snapshots reports -b bcfg2client
Bad entries:
 Package:nscd
 Package:cupsys
 File:/etc/ldap.conf
```

List extra entries for a single host:

```
$ bcfg2-admin snapshots reports -e bcfg2client
Extra entries:
 Package:python-pyxattr
 Package:librsync1
 Package:python-pylibacl
 Package:gcc-4.2-multilib
 Package:nxlibs
 Package:freenx-session-launcher
 Package:dx-doc
 Package:dirdiff
 Package:libhdf4g
 Package:nxclient
 Package:freenx-rdp
 Package:freenx-vnc
 Package:libxml2-dev
 Package:mysql-client
 Package:mysql-client-5.0
 Package:libxcompext3
 Package:lib32gomp1
 Package:dx
 Package:freenx-media
 Package:dxsamples
 Package:gcc-multilib
 Package:rdiff-backup
 Package:libdbd-mysql-perl
 Package:libxcomp3
 Package:freenx-server
 Package:smbfs
 Package:planner
 Package:nxagent
 Package:libc6-dev-i386
 Package:libfltk1.1-dev
 Package:freenx
 Package:libdx4
 Package:libxcompshad3
 Service:freenx-server
```

Detailed view of hosts for a particular date:

```
$ bcfg2-admin snapshots reports --date 2009 5 30
============= ========= ========================================== ========================
Client        Correct   Revision                                   Time
============= ========= ========================================== ========================
bcfg2client   False     10c1a12c62c57c0861cc453b8d2640c4839a7357   2009-05-29 10:52:34.7010
```

5.5.4 TODO/Wishlist

- Identify per-client changes in correctness over time
- Detailed view for a particular date
- Track entry changes over time (glibc updated on these dates to these versions)

CHAPTER

SIX

THE BCFG2 CLIENT

The Bcfg2 client attempts to reconcile the current configuration state with the configuration passed down from the server using various client tools. It does not perform any processing of the target configuration description. We chose this architecture, as opposed to one with a smarter client, for a few reasons:

- Client failure forces administrators to perform an O(n) reconfiguration operation. Simpler code is easier to debug and maintain.
- Minimize the bootstrap size; a complicated client can require more aspects of the system to function in order for reconfiguration to work.
- Isolate configuration generation functionality on the server, where it can be readily observed. This is the most complicated task that Bcfg2 performs.
- The results of the configuration process fit a fairly simple model. We wanted to validate it. The result is that Bcfg2 has a programmable deployment engine that can be driven by anything that writes a compatible configuration description.

6.1 Available client tools

6.1.1 Actions

This page describes use of the Action configuration entry. Action entries are commands that are executed either before bundle installation, after bundle installation or both. If exit status is observed, a failing pre-action will cause no modification of the enclosing bundle to be performed; all entries in included in that bundle will not be modified. Failing actions are reported through Bcfg2's reporting system, so they can be centrally observed. Actions look like:

```
<Action timing='pre|post|both'
        name='name'
        command='cmd text'
        when='always|modified'
        status='ignore|check'/>
```

Attribute	Values	Meaning
timing	pre, post, both	When the action is run
name	freeform	action name
command	freeform	command text
when	always, modified	If the action is always run, or only when a bundle should be or has been modified
status	ignore, check	If the return code of the action should be reported or not

Note that the status attribute tells the bcfg2 client to ignore return status, causing failures to still not be centrally reported. If central reporting of action failure is desired, set this attribute to 'check'. Also note that Action entries included in Base will not be executed.

Actions cannot be completely defined inside of a bundle; they are a bound entry, much like Packages, Services or Paths. The Rules plugin can bind these entries. For example to include the above action in a bundle, first the Action entry must be included in the bundle:

```
<Bundle name='bundle_name'>
  ...
  <Action name='action_name'/>
</Bundle>
```

Then a corresponding entry must be included in the Rules directory, like:

```
<Rules priority='0'>
<Action timing='post' when='modified' name='action_name' command='/path/to/command arg1 ar
</Rules>
```

This allows different clients to get different actions as a part of the same bundle based on group membership.

Example Action (add APT keys)

This example will add the '0C5A2783' for aptitude. It is useful to run this during the client bootstrap process so that the proper keys are installed prior to the bcfg2 client trying to install a package which requires this key.

```
<Rules priority='0'>
  <Group name='ubuntu'>
    <Action timing='post' name='apt-key-update' command='apt-key adv --recv-keys --keyserve
  </Group>
</Rules>
```

6.1.2 APT Client Tool

The APT tool allows you to configure custom options in `bcfg2.conf` for systems where the tools reside in non-standard locations. The available options (and their corresponding default values) are:

```
[APT]
install_path = '/usr'
var_path = '/var'
etc_path = '/etc'
```

6.1.3 Blast

Blastwave Packages. This tool driver is for blastwave packages on solaris.

6.1.4 Chkconfig

Tool to manage services (primarily on Red Hat based distros).

Note: Start and stop are standard arguments, but the one for reload isn't consistent across services. You can specify which argument to use with the *restart* property in Service tags. Example: `<Service name="ftp" restart="condrestart" status="on" type="chkconfig">`

6.1.5 DebInit

Debian Service Support; exec's update-rc.d to configure services.

6.1.6 Encap

Encap Packages.

6.1.7 FreeBSDInit

FreeBSD Service Support. Only bundle updates will work.

6.1.8 FreeBSDPackage

FreeBSD Packages. Verifies packages and their version numbers but can't install packages.

6.1.9 Available client tools

Client tool drivers allow Bcfg2 to execute configuration operations by interfacing with platform and distribution specific tools.

Tool drivers handle any reconfiguration or verification operation. So far we have tools that primarily deal with packaging systems and service management. The POSIX tool also handles file system and permissions/groups operations. To write your own tool driver, to handle a new packaging format, or new service architecture see *Writing A Client Tool Driver*.

When the Bcfg2 client is run, it attempts to instantiate each of these drivers. The succeeding list of drivers are printed as a debug message after this process has completed. Drivers can supercede one another, for example, the Yum driver conflicts (and unloads) the RPM driver. This behavior can be overridden by running the Bcfg2 client with the `-D` flag. This flag takes a colon delimited list of drivers to use on the system.

Currently these are the tool drivers that are distributed with Bcfg2:

Actions

This page describes use of the Action configuration entry. Action entries are commands that are executed either before bundle installation, after bundle installation or both. If exit status is observed, a failing pre-action will cause no modification of the enclosing bundle to be performed; all entries in included in that bundle will not be modified. Failing actions are reported through Bcfg2's reporting system, so they can be centrally observed. Actions look like:

```
<Action timing='pre|post|both'
        name='name'
        command='cmd text'
        when='always|modified'
        status='ignore|check'/>
```

Attribute	Values	Meaning
timing	pre, post, both	When the action is run
name	freeform	action name
command	freeform	command text
when	always, modified	If the action is always run, or only when a bundle should be or has been modified
status	ignore, check	If the return code of the action should be reported or not

Note that the status attribute tells the bcfg2 client to ignore return status, causing failures to still not be centrally reported. If central reporting of action failure is desired, set this attribute to 'check'. Also note that Action entries included in Base will not be executed.

Actions cannot be completely defined inside of a bundle; they are a bound entry, much like Packages, Services or Paths. The Rules plugin can bind these entries. For example to include the above action in a bundle, first the Action entry must be included in the bundle:

```
<Bundle name='bundle_name'>
  ...
  <Action name='action_name'/>
</Bundle>
```

Then a corresponding entry must be included in the Rules directory, like:

```
<Rules priority='0'>
<Action timing='post' when='modified' name='action_name' command='/path/to/command arg1 ar
</Rules>
```

This allows different clients to get different actions as a part of the same bundle based on group membership.

Example Action (add APT keys)

This example will add the '0C5A2783' for aptitude. It is useful to run this during the client bootstrap process so that the proper keys are installed prior to the bcfg2 client trying to install a package which requires this key.

```
<Rules priority='0'>
  <Group name='ubuntu'>
    <Action timing='post' name='apt-key-update' command='apt-key adv --recv-keys --keyserve
  </Group>
</Rules>
```

APT Client Tool

The APT tool allows you to configure custom options in `bcfg2.conf` for systems where the tools reside in non-standard locations. The available options (and their corresponding default values) are:

```
[APT]
install_path = '/usr'
var_path = '/var'
etc_path = '/etc'
```

Blast

Blastwave Packages. This tool driver is for blastwave packages on solaris.

Chkconfig

Tool to manage services (primarily on Red Hat based distros).

Note: Start and stop are standard arguments, but the one for reload isn't consistent across services. You can specify which argument to use with the *restart* property in Service tags. Example: `<Service name="ftp" restart="condrestart" status="on" type="chkconfig">`

DebInit

Debian Service Support; exec's update-rc.d to configure services.

Encap

Encap Packages.

FreeBSDInit

FreeBSD Service Support. Only bundle updates will work.

FreeBSDPackage

FreeBSD Packages. Verifies packages and their version numbers but can't install packages.

launchd

Mac OS X Services. To use this tool, you must maintain a standard launch daemon .plist file in `/Library/LaunchDaemons/` (example ssh.plist) and setup a `<Service name="com.openssh.sshd" type="launchd" status="on" />` entry in your config to load or unload the service. Note the name is the ''Label" specified inside of the .plist file

Portage

Support for Gentoo Packages.

POSIX

Files and Permissions are handled by the POSIX driver. Usage well documented other places.

RcUpdate

Uses the rc-update executable to manage services on distributions such as Gentoo.

RPM

> **Warning:** Deprecated in favor of *RPMng*

Executes rpm to manage packages most often on redhat based systems.

RPMng

Next-generation RPM tool, will be default in upcoming release. Handles RPM sublties like epoch and prelinking and 64-bit platforms better than RPM client tool.

SMF

Solaris Service Support.

Example legacy run service (lrc):

```
<BoundService name='/etc/rc2_d/S47pppd' FMRI='lrc:/etc/rc2_d/S47pppd' status='off' type='sr
```

SYSV

Handles System V Packaging format that is available on Solaris.

Upstart

Upstart service support. Uses Upstart to configure services.

Yum

Warning: Deprecated in favor of *YUMng*

Handles RPMs using the YUM package manager.

Bcfg2 RPMng/YUMng Client Drivers

Introduction

The goal of this driver is to resolve the issues that exist with the RPM and Yum client tool drivers.

For the most part, the issues are due to RPM being able to have multiple packages of the same name installed. This is an issue on all Red Hat and SUSE based distributions.

Examples of this are:

- SLES10 and openSUSE 10.2 both install six GPG keys. From an RPM perspective this means that there are six packages with the name gpg-pubkey.
- YUM always installs, as opposed to upgrades, kernel packages. This is hard coded in YUM (actually it can be overridden in yum.conf), so systems using YUM will eventually have multiple kernel packages installed.
- Red Hat family x86_64 based systems frequently have both an x86_64 and an i386 version of the same package installed.

The new Pkgmgr format files with Instances are therefore the only way to accurately describe an RPM based system. It is recommended that all RPM based systems be changed to use the new format configuration files and the RPMng driver. Alternatively, you can use the newer *Packages* plugin.

Development Status

Initial development of the drivers was done on Centos 4.4 x86_64, with testing on openSUSE 10.2 x86_64. Centos has been tested with a new style Pkgmgr file and openSUSE with an old style file (see the Configuration section below for what this means). Testing has now moved to Centos 5 x86_64 and old style files are no longer being tested.

RPMng/YUMng are the default RPM drivers.

Features

- Limited support for 0.9.4 and earlier Pkgmgr configuration files. See Configuration below for details.

- Full RPM package identification using epoch, version, release and arch.
- Support for multiple instances of packages with the Instance tag.
- Better control of the RPM verification using the pkg_checks, pkg_verify and verify_flags attributes.
- Support for install only packages such as the kernel packages.
- Support for per instance ignoring of individual files for the RPM verification with the Ignore tag.
- Multiple package Instances with full version information listed in interactive mode.
- Support for installation and removal of gpg-pubkey packages.
- Support for controlling what action is taken on package verification failure with the install_action, version_fail_action and verify_fail_action attributes.

RPMng Driver Overview

The RPMng driver uses a mixture of rpm commands and rpm-python as detailed in the sections below.

rpmtools module The rpmtools module conatins most of the rpm-python code and is imported by RPMng.py and YUMng.py.

RPMng.RefreshPackages() The RPMng.RefreshPackages method generates the installed dict using rpm-python code from the rpmtools module. Full name, epoch, version, release and arch information is stored.

RPMng.VerifyPackages() The RPMng.VerifyPackages method generates a number of structures that record the state of the of the system compared to the Bcfg2 literal configuration retrieved from the server. These structures are mainly used by the RPMng.Install method.

AS part of the verification process an rpm package level verification is carried out using rpm-python code from the rpmtools module. Full details of the failures are returned in a complicated dict/list structure for later use.

RPMng.Install() The RPMng.Install method attempts to fix what the RPMng.VerifyPackages method found wrong. It does this by installing, reinstalling, deleting and upgrading RPMs. RPMng.Install does not use rpm-python. It does use the following rppm commands as appropriate:

```
rpm -install

rpm --import

rpm -upgrade
```

A method (RPMng.to reinstall_check()) to decide whether to do a reinstall of a package instance or not has been added, but is very simple at this stage. Currently it will prevent a reinstall if the only reason for a verification failure was due to an RPM configuration (%config) file. A package reinstall will not replace these, so there is no point reinstalling.

RPMng.Remove() The RPMng.Remove method is written using rpm-python code in the rpmtools module. Full nevra information is used in the selection of the package removal.

Installation

isprelink This is a Python C extension module that checks to see if a file has been prelinked or not. It should be built and installed on systems that have the prelink package installed (only Red Hat family systems as far as I can tell). rpmtools will function without the isprelink module, but performance is not good.

Source can be found here ftp://ftp.mcs.anl.gov/pub/bcfg/isprelink-0.1.2.tar.gz

To compile and install prelink, execute:

```
python setup.py install
```

in the rpmtools directory. The elfutils-libelf-devel package is required for the compilation.

There are Centos x86_64 RPMs here ftp://ftp.mcs.anl.gov/pub/bcfg/redhat/

Configuration and Usage

Loading of RPMng The RPMng driver can be loaded by command line options, client configuration file options or as the default driver for RPM packages.

From the command line:

```
bcfg2 -n -v -d -D Action,POSIX,Chkconfig,RPMng
```

This produces quite a bit of output so you may want to redirect the output to a file for review.

In the `bcfg2.conf` file:

```
[client]
#drivers = Action,Chkconfig,POSIX,YUMng
drivers = Action,Chkconfig,POSIX,RPMng
```

Note: Note that loading this driver will unload the RPM driver, so the Yum driver will not work.

Configuration File Options A number of paramters can be set in the client configuration for both the RPMng and YUMng drivers. Each driver has its own section. A full client configuration file with all the options specified is below:

```
[communication]
protocol = xmlrpc/ssl
password = xxxxxx
user = yyyyyyy

[components]
bcfg2 = https://bcfg2:6789

[client]
```

```
#drivers = Action,Chkconfig,POSIX,YUMng
drivers = Action,Chkconfig,POSIX,RPMng

[RPMng]
pkg_checks = true
pkg_verify = true
erase_flags = allmatches
installonlypackages = kernel, kernel-bigmem, kernel-enterprise, kernel-smp, kernel-modules,
install_action = install
version_fail_action = upgrade
verify_fail_action = reinstall

[YUMng]
pkg_checks = True
pkg_verify = true
erase_flags = allmatches
autodep = true
installonlypackages = kernel, kernel-bigmem, kernel-enterprise, kernel-smp, kernel-modules,
install_action = install
version_fail_action = upgrade
verify_fail_action = reinstall
```

installOnlyPkgs Install only packages are packages that should only ever be installed or deleted, not upgraded.

The only packages for which this is an absolute on, are the gpg-pubkey packages. It is however 'best' practice to only ever install/delete kernel packages. The wisdom being that the package for the currently running kernel should always be installed. Doing an upgrade would delete the running kernel package.

The RPMng driver follows the YUM practice of having a list of install only packages. A default list is hard coded in RPMng.py. This maybe over ridden in the client configuration file.

Note that except for gpg-pubkey packages (which are always added to the list by the driver) the list in the client configuration file completely replaces the default list. An empty list means that there are no install only packages (except for gpg-pubkey), which is the behaviour of the old RPM driver.

Example - an empty list:

```
[RPMng]
installonlypackages =
```

Example - The default list:

```
[RPMng]
installonlypackages = kernel, kernel-bigmem, kernel-enterprise, kernel-smp, kernel-modules,
```

erase_flags erase_flags are rpm options used by 'rpm -erase' in the client Remove() method. The RPMng erase is written using rpm-python and does not use the rpm command.

The erase flags are specified in the client configuration file as a comma separated list and apply to all RPM erase operations. The default is:

```
[RPMng]
erase_flags = allmatches
```

The following rpm erase options are supported, see the rpm man page for details.:

```
noscripts
notriggers
repackage
allmatches
nodeps
```

Note: Note that specifying erase_flags in the configuration file completely replaces the default.

pkg_checks The RPMng/YUMng drivers do the following three checks/status:

1. Installed
2. Version
3. rpm verify

Setting pkg_checks = true (the default) in the client configuration file means that all three checks will be done for all packages.

Setting pkg_checks = false in the client configuration file means that only the Installed check will be done for all packages.

The true/false value can be any combination of upper and lower case.

Note:

1. pkg_checks must evaluate true for both the client (this option) and the package (see the Package Tag pkg_checks attribute below) for the action to take place.
2. If pkg_checks = false then the Pkgmgr entries do not need the version information. See the examples towards the bottom of the page.

pkg_verify The RPMng/YUMng drivers do the following three checks/status:

1. Installed
2. Version
3. rpm verify

Setting pkg_verify = true (the default) in the client configuration file means that all three checks will be done for all packages as long as pkg_checks = true.

Setting pkg_verify = false in the client configuration file means that the rpm verify wil not be done for all packages on the client.

The true/false value can be any combination of upper and lower case.

Note:

1. pkg_verify must evaluate true for both the client (this option) and the package instance (see the Instance Tag pkg_verify attribute below) for the action to take place.

install_action The RPMng/YUMng drivers do the following three checks/status:

1. Installed
2. Version
3. rpm verify

install_action controls whether or not a package instance will be installed if the installed check fails (i.e. if the package instance isn't installed).

If install_action = install then the package instance is installed. If install_action = none then the package instance is not installed.

Note:

1. install_action must evaluate true for both the client (this option) and the package instance (see the Instance Tag install_action attribute below) for the action to take place.

version_fail_action The RPMng/YUMng drivers do the following three checks/status:

1. Installed
2. Version
3. rpm verify

version_fail_action controls whether or not a package instance will be updated if the version check fails (i.e. if the installed package instance isn't the same version as specified in the configuration).

If version_fail_action = upgrade then the package instance is upgraded (or downgraded).

If version_fail_action = none then the package instance is not upgraded (or downgraded).

Note:

1. verion_fail_action must evaluate true for both the client (this option) and the package instance (see the Instance Tag version_fail_action attribute below) for the action to take place.

verify_fail_action The RPMng/YUMng drivers do the following three checks/status:

1. Installed
2. Version
3. rpm verify

verify_fail_action controls whether or not a package instance will be reinstlled if the version check fails (i.e. if the installed package instance isn't the same version as specified in the configuration).

If verify_fail_action = reinstall then the package instance is reinstalled. If verify_fail_action = none then the package instance is not reinstalled.

Note:

1. verify_fail_action must evaluate true for both the client (this option) and the package instance (see the Instance Tag verify_fail_action attribute below) for the action to take place.
2. yum cannot reinstall packages, so this option is really only relevant to RPMng.
3. RPMng will not attempt to reinstall a package instance if the only failure is an RPM configuration file.
4. RPMng will not attempt to reinstall a package instance if the only failure is an RPM dependency failure.

Interactive Mode Running the client in interactive mode (-I) prompts for the actions to be taken as before. Prompts are per package and may apply to multiple instances of that package. Each per package prompt will contain a list of actions per instance.

Actions are encoded as

D - Delete

I - Install

R - Reinstall

U - Upgrade/Downgrade

An example is below. The example is from a system that is still using the old Pkgmgr format, so the epoch and arch appear as '*'.:

```
Install/Upgrade/delete Package aaa_base instance(s) - R(*:10.2-38.*)  (y/N)
Install/Upgrade/delete Package evms instance(s) - R(*:2.5.5-67.*)  (y/N)
Install/Upgrade/delete Package gpg-pubkey instance(s) - D(*:9c800aca-40d8063e.*) D(*:0dfb3
Install/Upgrade/delete Package module-init-tools instance(s) - R(*:3.2.2-62.*)  (y/N)
Install/Upgrade/delete Package multipath-tools instance(s) - R(*:0.4.7-29.*)  (y/N)
Install/Upgrade/delete Package pam instance(s) - R(*:0.99.6.3-29.1.*)  (y/N)
Install/Upgrade/delete Package perl-AppConfig instance(s) - U(None:1.52-4.noarch -> *:1.63-
Install/Upgrade/delete Package postfix instance(s) - R(*:2.3.2-28.*)  (y/N)
Install/Upgrade/delete Package sysconfig instance(s) - R(*:0.60.4-3.*)  (y/N)
Install/Upgrade/delete Package udev instance(s) - R(*:103-12.*)  (y/N)
```

GPG Keys GPG is used by RPM to 'sign' packages. All vendor packages are signed with the vendors GPG key. Additional signatures maybe added to the rpm file at the users discretion.

It is normal to have multiple GPG keys installed. For example, SLES10 out of the box has six GPG keys installed.

To the RPM database all GPG 'packages' have the name 'gpg-pubkey', which may be nothing like the name of the file specified in the rpm -import command. For example on Centos 4 the file name is RPM-GPG-KEY-centos4. For SLES10 this means that there are six packages with the name 'gpg-pubkey' installed.

RPM does not check GPG keys at package installation, YUM does.

RPMng uses the rpm command for installation and does not therefore check GPG signatures at package install time. RPMng uses rpm-python for verification and does by default do signature checks as part of the

client Inventory process. To do the signature check the appropriate GPG keys must be installed. rpm-python is not very friendly if the required key(s) is not installed (it crashes the client).

The RPMng driver detects, on a per package instance basis, if the appropriate key is installed. If it is not, a warning message is printed and the signature check is disabled for that package instance, for that client run only.

GPG keys can be installed and removed by the RPMng driver. To install a GPG key configure it in Pkgmgr/Rules as a package and add gpg-pubkey to the clients abstract configuration. The gpg-pubkey package/instance is treated as an install only package. gpg-pubkey packages are installed by the RPMng driver with the rpm -import command.

gpg-pubkey packages will be removed by `bcfg2 -r packages` if they are not in the clients configuration.

```
<PackageList uri='http://fortress/' priority='0' type='rpm'>
    <Group name='Centos4.4-Standard'>
        <Group name='x86_64'>
            <Package name='gpg-pubkey' type='rpm'>
                <Instance simplefile='mrepo/Centos44-x86_64/disc1/RPM-GPG-KEY-centos4' vers
                <Instance simplefile='RPM-GPG-KEY-mbrady' version='9c777da4' release='4515
            </Package>
        </Group>
    </Group>
</PackageList>
```

Example gpg-pubkey Pkgmgr configuration file.

Pkgmgr Configuration Also see the general *Pkgmgr* and *Fun and Profit using altsrc* pages.

Package Tag (Old style) Old style (meaning no Instance tag) Pkgmgr files have limited support. Specifically the multiarch and verify attributes are ignored.

If multiarch type support is needed a new style format file must be used.

If some control over the verification is needed, replace the verify attribute with the pkg_checks or verify_flags attributes. The pkg_checks and verify_flags attributes are detailed under the Instance tag heading.

Package Tag (New Style) and Attributes The new style package tag supports the name and pkg_checks attributes and requires the use of Instance tag entries.

New style configuration files must be generated from the RPM headers. Either from RPM files or from the RPM DB.

The included pkgmgr_gen.py can be used as a starting point for generating configuration files from directories of RPM package files. pkgmgr_gen.py –help for the options.

The included pkgmgr_update.py can be used to update the package instance versions in configuration files from directories of package files. pkgmgr_update.py –help for the options.

Attribute	Description	Values
name	Package name.	String
pkg_checks	Do the version and rpm verify checks.	true(default) or false

Instance Tag and Attributes The instance tag supports the following attributes:

Attribute	Description	Values
simplefile	Package file name.	String (see Notes below)
epoch	Package epoch.	String (numeric only) (optional)
version	Package version.	String
release	Package release.	String
arch	Package architecture.	Architecture String e.g. (i386|i586|i686|x86_64)
verify_flags	Comma separated list of rpm –verify options. See the rpm man page for their details.	nodeps, nodigest, nofiles, noscripts, nosignature, nolinkto, nomd5, nosize, nouser, nogroup, nomtime, nomode, nordev
pkg_verify	Do the rpm verify	true(default) or false
install_action	Install package instance if it is not installed.	install(default) or none
version_fail_action	Upgrade package if the incorrect version is installed.	upgrade(default) or none
verify_fail_action	Reinstall the package instance if the rpm verify failed	reinstall(default) or none

Note: The simplefile attribute doesn't need to be just the filename, meaning the basename. It is joined with the uri attribute from the PackageList Tag to form the URL that the client will use to download the package. So the uri could just be the host portion of the url and simple file could be the directory path.

e.g.

```
<PackageList uri='http://fortress/' priority='0' type='rpm'>
    <Group name='Centos4.4-Standard'>
        <Group name='x86_64'>
            <Package name='gpg-pubkey' type='rpm'>
                <Instance simplefile='mrepo/Centos44-x86_64/disc1/RPM-GPG-KEY-centos4' vers
                <Instance simplefile='RPM-GPG-KEY-mbrady' version='9c777da4' release='4515
            </Package>
        </Group>
    </Group>
</PackageList>
```

The values for epoch, version, release and arch attributes must come from the RPM header, not the RPM file name.

Epoch is a strange thing. In short:

```
epoch not set == epoch=None < epoch='0' < epoch='1'
```

and it is an int, but elementtree attributes have to be str or unicode, so the driver is constantly converting.

Ignore Tag The Ignore tag is used to "mask out" individual files from the RPM verification. This is done by comparing the verification failure results with the Ignore tag name. If there is a match, that entry is not used by the client to determine if a package has failed verification.

Ignore tag entries can be specified at both the Package level, in which case they apply to all Instances, and/or at the Instance level, in which case they only apply to that instance.

Ignore tag entries are used by the RPMng driver. They can be specified in both old and new style Pkgmgr files.

The Ignore Tag supports the following attributes:

Attribute	Description	Values
name	File name.	String

Example

```
<Package name='glibc' type='rpm'>
    <Ignore name='/etc/rpc'/>
    <Instance simplefile='glibc-2.3.4-2.25.x86_64.rpm' version='2.3.4' release='2.25' arch=
</Package>
```

POSIX 'ignore' Path entries The YUMng analog to the Ignore Tag used by RPMng is the use of Path entries of type 'ignore'. The following shows an example for the centos-release package which doesn't verify if you remove the default repos and replace them with a custom repo.

```
<!-- Ignore verification failures for centos-release -->
<BoundPath name='/etc/yum.repos.d/CentOS-Base.repo' type='ignore'/>
<BoundPath name='/etc/yum.repos.d/CentOS-Media.repo' type='ignore'/>
```

Automated Generation of Pkgmgr Configuration Files The two utilities detailed below are provided in the tools directory of the source tarball.

Also see the general *Pkgmgr* and *Fun and Profit using altsrc* pages.

pkgmgr_gen.py pkgmgr_gen will generate a Pkgmgr file from a list of directories containing RPMs or from a list of YUM repositories.:

```
[root@bcfg2 Pkgmgr]# pkgmgr_gen.py --help usage: pkgmgr_gen.py
[options]

options:
  -h, --help            show this help message and exit
  -aARCHS, --archs=ARCHS
                        Comma separated list of subarchitectures to include.
                        The highest subarichitecture required in an
                        architecture group should specified.   Lower
                        subarchitecture packages will be loaded if that
                        is all that is available. e.g. The higher of i386,
                        i486 and i586 packages will be loaded if -a i586
                        is specified. (Default: all).
```

```
  -dRPMDIRS, --rpmdirs=RPMDIRS
                        Comma separated list of directories to scan for RPMS.
                        Wilcards are permitted.
  -eENDDATE, --enddate=ENDDATE
                        End date for RPM file selection.
  -fFORMAT, --format=FORMAT
                        Format of the Output. Choices are yum or rpm.
                        (Default: yum)
  -gGROUPS, --groups=GROUPS
                        List of comma separated groups to nest Package
                        entities in.
  -iINDENT, --indent=INDENT
                        Number of leading spaces to indent nested entries in
                        the                                              output.
                        (Default:4)
  -oOUTFILE, --outfile=OUTFILE
                        Output file name.
  -P, --pkgmgrhdr       Include PackageList header in output.
  -pPRIORITY, --priority=PRIORITY
                        Value to set priority attribute in the PackageList Tag.
                        (Default: 0)
  -rRELEASE, --release=RELEASE
                        Which releases to include in the output. Choices are
                        all or latest.  (Default: latest).
  -sSTARTDATE, --startdate=STARTDATE
                        Start date for RPM file selection.
  -uURI, --uri=URI      URI for PackageList header required for RPM format
                        ouput.
  -v, --verbose         Enable verbose output.
  -yYUMREPOS, --yumrepos=YUMREPOS
                        Comma separated list of YUM repository URLs to load.
                        NOTE: Each URL must end in a '/' character.
```

Note: The startdate and enddate options are not yet implemented.

pkgmgr_update.py pkgmgr_update will update the release (meaning the epoch, version and release) information in an existing Pkgrmgr file from a list of directories containing RPMs or from a list of YUM repositories. All Tags and other attributes in the existing file will remain unchanged.:

```
[root@bcfg2 Pkgmgr]# pkgmgr_update.py --help
usage: pkgmgr_update.py [options]

options:
  -h, --help            show this help message and exit
  -cCONFIGFILE, --configfile=CONFIGFILE
                        Existing Pkgmgr configuration  file name.
  -dRPMDIRS, --rpmdirs=RPMDIRS
                        Comma separated list of directories to scan for RPMS.
                        Wilcards are permitted.
  -oOUTFILE, --outfile=OUTFILE
                        Output file name or new Pkgrmgr file.
  -v, --verbose         Enable verbose output.
```

```
  -yYUMREPOS, --yumrepos=YUMREPOS
                        Comma separated list of YUM repository URLs to load.
                        NOTE: Each URL must end in a '/' character.
```

Pkgmgr Configuration Examples

verify_flags This entry was used for the Centos test client used during RPMng development.

```
<Package name='bcfg2' type='rpm'>
    <Instance simplefile='bcfg2-0.9.3-0.0pre5.noarch.rpm' version='0.9.3' release='0.0pre5'
</Package>
```

Multiple Instances

```
<Package name='beecrypt' type='rpm'>
    <Instance simplefile='beecrypt-3.1.0-6.x86_64.rpm' version='3.1.0' release='6' arch='x8
    <Instance simplefile='beecrypt-3.1.0-6.i386.rpm' version='3.1.0' release='6' arch='i386
</Package>
```

Kernel **Note:** Multiple instances with the same architecture must be in the installOnlyPkgs list.

```
<Package name='kernel' type='rpm'>
    <Instance simplefile='kernel-2.6.9-42.0.8.EL.x86_64.rpm' version='2.6.9' release='42.0
    <Instance simplefile='kernel-2.6.9-42.0.10.EL.x86_64.rpm' version='2.6.9' release='42.0
</Package>
```

Per Instance Ignore **Note:** In this case a per instance ignore is actually a bad idea as the verify failure is because of multiarch issues where the last package installed wins. So this would be better as a Package level ignore.

Ignore tag entries only work with the RPMng driver. They do not appear to be supported in YUMng as of 1.0pre5.

```
<Package name='glibc' type='rpm'>
    <Instance simplefile='glibc-2.3.4-2.25.x86_64.rpm' version='2.3.4' release='2.25' arch=
        <Ignore name='/etc/rpc'/>
    </Instance>
    <Instance simplefile='glibc-2.3.4-2.25.i686.rpm' version='2.3.4' release='2.25' arch='i
</Package>
```

pkg_checks If pkg_checks = false the version information is not required. If pkg_checks = true the full information is needed as normal.

For YUMng a minimal entry is

```
<Package name="bcfg2" type="yum" pkg_checks="False"/>
```

In fact for YUMng, with pkg_checks = false, any combination of the nevra attributes that will build a valid yum package name (see the Misc heading on the yum man page) is valid.

```
<Package name="bcfg2" type="yum" pkg_checks="False" arch="x86_64"/>
```

For RPMng a minimal entry is

```
<Package name="bcfg2" type="rpm" pkg_checks="False" simplefile="bcfg2-0.9.4-0.0pre1.noarch
```

verify_fail_action The way I have Bcfg2 configured for my development systems. This way it reports bad, but doesn't do anything about it.

```
<Package name='bcfg2' type='rpm'>
    <Instance simplefile='bcfg2-0.9.3-0.0pre5.noarch.rpm' version='0.9.3' release='0.0pre5'
</Package>
```

6.1.10 launchd

Mac OS X Services. To use this tool, you must maintain a standard launch daemon .plist file in `/Library/LaunchDaemons/` (example ssh.plist) and setup a `<Service name="com.openssh.sshd" type="launchd" status="on" />` entry in your config to load or unload the service. Note the name is the ''Label" specified inside of the .plist file

6.1.11 Portage

Support for Gentoo Packages.

6.1.12 POSIX

Files and Permissions are handled by the POSIX driver. Usage well documented other places.

6.1.13 RcUpdate

Uses the rc-update executable to manage services on distributions such as Gentoo.

6.1.14 RPM

> **Warning:** Deprecated in favor of *RPMng*

Executes rpm to manage packages most often on redhat based systems.

6.1.15 RPMng

Next-generation RPM tool, will be default in upcoming release. Handles RPM sublties like epoch and prelinking and 64-bit platforms better than RPM client tool.

6.1.16 SMF

Solaris Service Support.

Example legacy run service (lrc):

```
<BoundService name='/etc/rc2_d/S47pppd' FMRI='lrc:/etc/rc2_d/S47pppd' status='off' type='sr
```

6.1.17 SYSV

Handles System V Packaging format that is available on Solaris.

6.1.18 Upstart

Upstart service support. Uses Upstart to configure services.

6.1.19 Yum

Warning: Deprecated in favor of *YUMng*

Handles RPMs using the YUM package manager.

6.1.20 Bcfg2 RPMng/YUMng Client Drivers

Introduction

The goal of this driver is to resolve the issues that exist with the RPM and Yum client tool drivers.

For the most part, the issues are due to RPM being able to have multiple packages of the same name installed. This is an issue on all Red Hat and SUSE based distributions.

Examples of this are:

- SLES10 and openSUSE 10.2 both install six GPG keys. From an RPM perspective this means that there are six packages with the name gpg-pubkey.
- YUM always installs, as opposed to upgrades, kernel packages. This is hard coded in YUM (actually it can be overridden in yum.conf), so systems using YUM will eventually have multiple kernel packages installed.

- Red Hat family x86_64 based systems frequently have both an x86_64 and an i386 version of the same package installed.

The new Pkgmgr format files with Instances are therefore the only way to accurately describe an RPM based system. It is recommended that all RPM based systems be changed to use the new format configuration files and the RPMng driver. Alternatively, you can use the newer *Packages* plugin.

Development Status

Initial development of the drivers was done on Centos 4.4 x86_64, with testing on openSUSE 10.2 x86_64. Centos has been tested with a new style Pkgmgr file and openSUSE with an old style file (see the Configuration section below for what this means). Testing has now moved to Centos 5 x86_64 and old style files are no longer being tested.

RPMng/YUMng are the default RPM drivers.

Features

- Limited support for 0.9.4 and earlier Pkgmgr configuration files. See Configuration below for details.
- Full RPM package identification using epoch, version, release and arch.
- Support for multiple instances of packages with the Instance tag.
- Better control of the RPM verification using the pkg_checks, pkg_verify and verify_flags attributes.
- Support for install only packages such as the kernel packages.
- Support for per instance ignoring of individual files for the RPM verification with the Ignore tag.
- Multiple package Instances with full version information listed in interactive mode.
- Support for installation and removal of gpg-pubkey packages.
- Support for controlling what action is taken on package verification failure with the install_action, version_fail_action and verify_fail_action attributes.

RPMng Driver Overview

The RPMng driver uses a mixture of rpm commands and rpm-python as detailed in the sections below.

rpmtools module

The rpmtools module conatins most of the rpm-python code and is imported by RPMng.py and YUMng.py.

RPMng.RefreshPackages()

The RPMng.RefreshPackages method generates the installed dict using rpm-python code from the rpmtools module. Full name, epoch, version, release and arch information is stored.

RPMng.VerifyPackages()

The RPMng.VerifyPackages method generates a number of structures that record the state of the of the system compared to the Bcfg2 literal configuration retrieved from the server. These structures are mainly used by the RPMng.Install method.

AS part of the verification process an rpm package level verification is carried out using rpm-python code from the rpmtools module. Full details of the failures are returned in a complicated dict/list structure for later use.

RPMng.Install()

The RPMng.Install method attempts to fix what the RPMng.VerifyPackages method found wrong. It does this by installing, reinstalling, deleting and upgrading RPMs. RPMng.Install does not use rpm-python. It does use the following rppm commands as appropriate:

```
rpm -install
```

```
rpm --import
```

```
rpm -upgrade
```

A method (RPMng.to reinstall_check()) to decide whether to do a reinstall of a package instance or not has been added, but is very simple at this stage. Currently it will prevent a reinstall if the only reason for a verification failure was due to an RPM configuration (%config) file. A package reinstall will not replace these, so there is no point reinstalling.

RPMng.Remove()

The RPMng.Remove method is written using rpm-python code in the rpmtools module. Full nevra information is used in the selection of the package removal.

Installation

isprelink

This is a Python C extension module that checks to see if a file has been prelinked or not. It should be built and installed on systems that have the prelink package installed (only Red Hat family systems as far as I can tell). rpmtools will function without the isprelink module, but performance is not good.

Source can be found here ftp://ftp.mcs.anl.gov/pub/bcfg/isprelink-0.1.2.tar.gz

To compile and install prelink, execute:

```
python setup.py install
```

in the rpmtools directory. The elfutils-libelf-devel package is required for the compilation.

There are Centos x86_64 RPMs here ftp://ftp.mcs.anl.gov/pub/bcfg/redhat/

Configuration and Usage

Loading of RPMng

The RPMng driver can be loaded by command line options, client configuration file options or as the default driver for RPM packages.

From the command line:

```
bcfg2 -n -v -d -D Action,POSIX,Chkconfig,RPMng
```

This produces quite a bit of output so you may want to redirect the output to a file for review.

In the `bcfg2.conf` file:

```
[client]
#drivers = Action,Chkconfig,POSIX,YUMng
drivers = Action,Chkconfig,POSIX,RPMng
```

Note: Note that loading this driver will unload the RPM driver, so the Yum driver will not work.

Configuration File Options

A number of paramters can be set in the client configuration for both the RPMng and YUMng drivers. Each driver has its own section. A full client configuration file with all the options specified is below:

```
[communication]
protocol = xmlrpc/ssl
password = xxxxxx
user = yyyyyyy

[components]
bcfg2 = https://bcfg2:6789

[client]
#drivers = Action,Chkconfig,POSIX,YUMng
drivers = Action,Chkconfig,POSIX,RPMng

[RPMng]
pkg_checks = true
pkg_verify = true
erase_flags = allmatches
installonlypackages = kernel, kernel-bigmem, kernel-enterprise, kernel-smp, kernel-modules,
install_action = install
version_fail_action = upgrade
verify_fail_action = reinstall

[YUMng]
pkg_checks = True
```

```
pkg_verify = true
erase_flags = allmatches
autodep = true
installonlypackages = kernel, kernel-bigmem, kernel-enterprise, kernel-smp, kernel-modules,
install_action = install
version_fail_action = upgrade
verify_fail_action = reinstall
```

installOnlyPkgs Install only packages are packages that should only ever be installed or deleted, not upgraded.

The only packages for which this is an absolute on, are the gpg-pubkey packages. It is however 'best' practice to only ever install/delete kernel packages. The wisdom being that the package for the currently running kernel should always be installed. Doing an upgrade would delete the running kernel package.

The RPMng driver follows the YUM practice of having a list of install only packages. A default list is hard coded in RPMng.py. This maybe over ridden in the client configuration file.

Note that except for gpg-pubkey packages (which are always added to the list by the driver) the list in the client configuration file completely replaces the default list. An empty list means that there are no install only packages (except for gpg-pubkey), which is the behaviour of the old RPM driver.

Example - an empty list:

```
[RPMng]
installonlypackages =
```

Example - The default list:

```
[RPMng]
installonlypackages = kernel, kernel-bigmem, kernel-enterprise, kernel-smp, kernel-modules,
```

erase_flags erase_flags are rpm options used by 'rpm -erase' in the client Remove() method. The RPMng erase is written using rpm-python and does not use the rpm command.

The erase flags are specified in the client configuration file as a comma separated list and apply to all RPM erase operations. The default is:

```
[RPMng]
erase_flags = allmatches
```

The following rpm erase options are supported, see the rpm man page for details.:

```
noscripts
notriggers
repackage
allmatches
nodeps
```

Note: Note that specifying erase_flags in the configuration file completely replaces the default.

pkg_checks The RPMng/YUMng drivers do the following three checks/status:

1. Installed
2. Version
3. rpm verify

Setting pkg_checks = true (the default) in the client configuration file means that all three checks will be done for all packages.

Setting pkg_checks = false in the client configuration file means that only the Installed check will be done for all packages.

The true/false value can be any combination of upper and lower case.

Note:

1. pkg_checks must evaluate true for both the client (this option) and the package (see the Package Tag pkg_checks attribute below) for the action to take place.
2. If pkg_checks = false then the Pkgmgr entries do not need the version information. See the examples towards the bottom of the page.

pkg_verify The RPMng/YUMng drivers do the following three checks/status:

1. Installed
2. Version
3. rpm verify

Setting pkg_verify = true (the default) in the client configuration file means that all three checks will be done for all packages as long as pkg_checks = true.

Setting pkg_verify = false in the client configuration file means that the rpm verify wil not be done for all packages on the client.

The true/false value can be any combination of upper and lower case.

Note:

1. pkg_verify must evaluate true for both the client (this option) and the package instance (see the Instance Tag pkg_verify attribute below) for the action to take place.

install_action The RPMng/YUMng drivers do the following three checks/status:

1. Installed
2. Version
3. rpm verify

install_action controls whether or not a package instance will be installed if the installed check fails (i.e. if the package instance isn't installed).

If install_action = install then the package instance is installed. If install_action = none then the package instance is not installed.

Note:

1. install_action must evaluate true for both the client (this option) and the package instance (see the Instance Tag install_action attribute below) for the action to take place.

version_fail_action The RPMng/YUMng drivers do the following three checks/status:

1. Installed
2. Version
3. rpm verify

version_fail_action controls whether or not a package instance will be updated if the version check fails (i.e. if the installed package instance isn't the same version as specified in the configuration).

If version_fail_action = upgrade then the package instance is upgraded (or downgraded).

If version_fail_action = none then the package instance is not upgraded (or downgraded).

Note:

1. verion_fail_action must evaluate true for both the client (this option) and the package instance (see the Instance Tag version_fail_action attribute below) for the action to take place.

verify_fail_action The RPMng/YUMng drivers do the following three checks/status:

1. Installed
2. Version
3. rpm verify

verify_fail_action controls whether or not a package instance will be reinstlled if the version check fails (i.e. if the installed package instance isn't the same version as specified in the configuration).

If verify_fail_action = reinstall then the package instance is reinstalled. If verify_fail_action = none then the package instance is not reinstalled.

Note:

1. verify_fail_action must evaluate true for both the client (this option) and the package instance (see the Instance Tag verify_fail_action attribute below) for the action to take place.
2. yum cannot reinstall packages, so this option is really only relevant to RPMng.
3. RPMng will not attempt to reinstall a package instance if the only failure is an RPM configuration file.
4. RPMng will not attempt to reinstall a package instance if the only failure is an RPM dependency failure.

Interactive Mode

Running the client in interactive mode (-I) prompts for the actions to be taken as before. Prompts are per package and may apply to multiple instances of that package. Each per package prompt will contain a list of actions per instance.

Actions are encoded as

D - Delete

I - Install

R - Reinstall

U - Upgrade/Downgrade

An example is below. The example is from a system that is still using the old Pkgmgr format, so the epoch and arch appear as '*'.:

```
Install/Upgrade/delete Package aaa_base instance(s) - R(*:10.2-38.*)  (y/N)
Install/Upgrade/delete Package evms instance(s) - R(*:2.5.5-67.*)  (y/N)
Install/Upgrade/delete Package gpg-pubkey instance(s) - D(*:9c800aca-40d8063e.*) D(*:0dfb3
Install/Upgrade/delete Package module-init-tools instance(s) - R(*:3.2.2-62.*)  (y/N)
Install/Upgrade/delete Package multipath-tools instance(s) - R(*:0.4.7-29.*)  (y/N)
Install/Upgrade/delete Package pam instance(s) - R(*:0.99.6.3-29.1.*)  (y/N)
Install/Upgrade/delete Package perl-AppConfig instance(s) - U(None:1.52-4.noarch -> *:1.63-
Install/Upgrade/delete Package postfix instance(s) - R(*:2.3.2-28.*)  (y/N)
Install/Upgrade/delete Package sysconfig instance(s) - R(*:0.60.4-3.*)  (y/N)
Install/Upgrade/delete Package udev instance(s) - R(*:103-12.*)  (y/N)
```

GPG Keys

GPG is used by RPM to 'sign' packages. All vendor packages are signed with the vendors GPG key. Additional signatures maybe added to the rpm file at the users discretion.

It is normal to have multiple GPG keys installed. For example, SLES10 out of the box has six GPG keys installed.

To the RPM database all GPG 'packages' have the name 'gpg-pubkey', which may be nothing like the name of the file specified in the rpm -import command. For example on Centos 4 the file name is RPM-GPG-KEY-centos4. For SLES10 this means that there are six packages with the name 'gpg-pubkey' installed.

RPM does not check GPG keys at package installation, YUM does.

RPMng uses the rpm command for installation and does not therefore check GPG signatures at package install time. RPMng uses rpm-python for verification and does by default do signature checks as part of the client Inventory process. To do the signature check the appropriate GPG keys must be installed. rpm-python is not very friendly if the required key(s) is not installed (it crashes the client).

The RPMng driver detects, on a per package instance basis, if the appropriate key is installed. If it is not, a warning message is printed and the signature check is disabled for that package instance, for that client run only.

GPG keys can be installed and removed by the RPMng driver. To install a GPG key configure it in Pkgmgr/Rules as a package and add gpg-pubkey to the clients abstract configuration. The gpg-pubkey package/instance is treated as an install only package. gpg-pubkey packages are installed by the RPMng driver with the rpm -import command.

gpg-pubkey packages will be removed by `bcfg2 -r packages` if they are not in the clients configuration.

```
<PackageList uri='http://fortress/' priority='0' type='rpm'>
    <Group name='Centos4.4-Standard'>
        <Group name='x86_64'>
            <Package name='gpg-pubkey' type='rpm'>
                <Instance simplefile='mrepo/Centos44-x86_64/disc1/RPM-GPG-KEY-centos4' vers
                <Instance simplefile='RPM-GPG-KEY-mbrady' version='9c777da4' release='4515
            </Package>
        </Group>
    </Group>
</PackageList>
```

Example gpg-pubkey Pkgmgr configuration file.

Pkgmgr Configuration

Also see the general *Pkgmgr* and *Fun and Profit using altsrc* pages.

Package Tag (Old style) Old style (meaning no Instance tag) Pkgmgr files have limited support. Specifically the multiarch and verify attributes are ignored.

If multiarch type support is needed a new style format file must be used.

If some control over the verification is needed, replace the verify attribute with the pkg_checks or verify_flags attributes. The pkg_checks and verify_flags attributes are detailed under the Instance tag heading.

Package Tag (New Style) and Attributes The new style package tag supports the name and pkg_checks attributes and requires the use of Instance tag entries.

New style configuration files must be generated from the RPM headers. Either from RPM files or from the RPM DB.

The included pkgmgr_gen.py can be used as a starting point for generating configuration files from directories of RPM package files. pkgmgr_gen.py –help for the options.

The included pkgmgr_update.py can be used to update the package instance versions in configuration files from directories of package files. pkgmgr_update.py –help for the options.

Attribute	Description	Values
name	Package name.	String
pkg_checks	Do the version and rpm verify checks.	true(default) or false

Instance Tag and Attributes The instance tag supports the following attributes:

Attribute	Description	Values
simplefile	Package file name.	String (see Notes below)
epoch	Package epoch.	String (numeric only) (optional)
version	Package version.	String
release	Package release.	String
arch	Package architecture.	Architecture String e.g. (i386|i586|i686|x86_64)
verify_flags	Comma separated list of rpm –verify options. See the rpm man page for their details.	nodeps, nodigest, nofiles, noscripts, nosignature, nolinkto, nomd5, nosize, nouser, nogroup, nomtime, nomode, nordev
pkg_verify	Do the rpm verify	true(default) or false
install_action	Install package instance if it is not installed.	install(default) or none
version_fail_action	Upgrade package if the incorrect version is installed.	upgrade(default) or none
verify_fail_action	Reinstall the package instance if the rpm verify failed	reinstall(default) or none

Note: The simplefile attribute doesn't need to be just the filename, meaning the basename. It is joined with the uri attribute from the PackageList Tag to form the URL that the client will use to download the package. So the uri could just be the host portion of the url and simple file could be the directory path.

e.g.

```
<PackageList uri='http://fortress/' priority='0' type='rpm'>
    <Group name='Centos4.4-Standard'>
        <Group name='x86_64'>
            <Package name='gpg-pubkey' type='rpm'>
                <Instance simplefile='mrepo/Centos44-x86_64/disc1/RPM-GPG-KEY-centos4' vers
                <Instance simplefile='RPM-GPG-KEY-mbrady' version='9c777da4' release='4515
            </Package>
        </Group>
    </Group>
</PackageList>
```

The values for epoch, version, release and arch attributes must come from the RPM header, not the RPM file name.

Epoch is a strange thing. In short:

```
epoch not set == epoch=None < epoch='0' < epoch='1'
```

and it is an int, but elementtree attributes have to be str or unicode, so the driver is constantly converting.

Ignore Tag The Ignore tag is used to "mask out" individual files from the RPM verification. This is done by comparing the verification failure results with the Ignore tag name. If there is a match, that entry is not used by the client to determine if a package has failed verification.

Ignore tag entries can be specified at both the Package level, in which case they apply to all Instances, and/or at the Instance level, in which case they only apply to that instance.

Ignore tag entries are used by the RPMng driver. They can be specified in both old and new style Pkgmgr files.

The Ignore Tag supports the following attributes:

Attribute	Description	Values
name	File name.	String

Example

```
<Package name='glibc' type='rpm'>
    <Ignore name='/etc/rpc'/>
    <Instance simplefile='glibc-2.3.4-2.25.x86_64.rpm' version='2.3.4' release='2.25' arch=
</Package>
```

POSIX 'ignore' Path entries The YUMng analog to the Ignore Tag used by RPMng is the use of Path entries of type 'ignore'. The following shows an example for the centos-release package which doesn't verify if you remove the default repos and replace them with a custom repo.

```
<!-- Ignore verification failures for centos-release -->
<BoundPath name='/etc/yum.repos.d/CentOS-Base.repo' type='ignore'/>
<BoundPath name='/etc/yum.repos.d/CentOS-Media.repo' type='ignore'/>
```

Automated Generation of Pkgmgr Configuration Files

The two utilities detailed below are provided in the tools directory of the source tarball.

Also see the general *Pkgmgr* and *Fun and Profit using altsrc* pages.

pkgmgr_gen.py pkgmgr_gen will generate a Pkgmgr file from a list of directories containing RPMs or from a list of YUM repositories.:

```
[root@bcfg2 Pkgmgr]# pkgmgr_gen.py --help usage: pkgmgr_gen.py
[options]

options:
  -h, --help            show this help message and exit
  -aARCHS, --archs=ARCHS
                        Comma separated list of subarchitectures to include.
                        The highest subarichitecture required in an
                        architecture group should specified.   Lower
                        subarchitecture packages will be loaded if that
                        is all that is available. e.g. The higher of i386,
                        i486 and i586 packages will be loaded if -a i586
                        is specified. (Default: all).
  -dRPMDIRS, --rpmdirs=RPMDIRS
                        Comma separated list of directories to scan for RPMS.
                        Wilcards are permitted.
```

```
  -eENDDATE, --enddate=ENDDATE
                        End date for RPM file selection.
  -fFORMAT, --format=FORMAT
                        Format of the Output. Choices are yum or rpm.
                        (Default: yum)
  -gGROUPS, --groups=GROUPS
                        List of comma separated groups to nest Package
                        entities in.
  -iINDENT, --indent=INDENT
                        Number of leading spaces to indent nested entries in
                        the                                              output.
                        (Default:4)
  -oOUTFILE, --outfile=OUTFILE
                        Output file name.
  -P, --pkgmgrhdr       Include PackageList header in output.
  -pPRIORITY, --priority=PRIORITY
                        Value to set priority attribute in the PackageList Tag.
                        (Default: 0)
  -rRELEASE, --release=RELEASE
                        Which releases to include in the output. Choices are
                        all or latest.  (Default: latest).
  -sSTARTDATE, --startdate=STARTDATE
                        Start date for RPM file selection.
  -uURI, --uri=URI      URI for PackageList header required for RPM format
                        ouput.
  -v, --verbose         Enable verbose output.
  -yYUMREPOS, --yumrepos=YUMREPOS
                        Comma separated list of YUM repository URLs to load.
                        NOTE: Each URL must end in a '/' character.
```

Note: The startdate and enddate options are not yet implemented.

pkgmgr_update.py

pkgmgr_update will update the release (meaning the epoch, version and release) information in an existing Pkgrmgr file from a list of directories containing RPMs or from a list of YUM repositories. All Tags and other attributes in the existing file will remain unchanged.:

```
[root@bcfg2 Pkgmgr]# pkgmgr_update.py --help
usage: pkgmgr_update.py [options]

options:
  -h, --help            show this help message and exit
  -cCONFIGFILE, --configfile=CONFIGFILE
                        Existing Pkgmgr configuration  file name.
  -dRPMDIRS, --rpmdirs=RPMDIRS
                        Comma separated list of directories to scan for RPMS.
                        Wilcards are permitted.
  -oOUTFILE, --outfile=OUTFILE
                        Output file name or new Pkgrmgr file.
  -v, --verbose         Enable verbose output.
  -yYUMREPOS, --yumrepos=YUMREPOS
```

```
                    Comma separated list of YUM repository URLs to load.
                    NOTE: Each URL must end in a '/' character.
```

Pkgmgr Configuration Examples

verify_flags This entry was used for the Centos test client used during RPMng development.

```
<Package name='bcfg2' type='rpm'>
    <Instance simplefile='bcfg2-0.9.3-0.0pre5.noarch.rpm' version='0.9.3' release='0.0pre5'
</Package>
```

Multiple Instances

```
<Package name='beecrypt' type='rpm'>
    <Instance simplefile='beecrypt-3.1.0-6.x86_64.rpm' version='3.1.0' release='6' arch='x8
    <Instance simplefile='beecrypt-3.1.0-6.i386.rpm' version='3.1.0' release='6' arch='i386
</Package>
```

Kernel **Note:** Multiple instances with the same architecture must be in the installOnlyPkgs list.

```
<Package name='kernel' type='rpm'>
    <Instance simplefile='kernel-2.6.9-42.0.8.EL.x86_64.rpm' version='2.6.9' release='42.0
    <Instance simplefile='kernel-2.6.9-42.0.10.EL.x86_64.rpm' version='2.6.9' release='42.0
</Package>
```

Per Instance Ignore **Note:** In this case a per instance ignore is actually a bad idea as the verify failure is because of multiarch issues where the last package installed wins. So this would be better as a Package level ignore.

Ignore tag entries only work with the RPMng driver. They do not appear to be supported in YUMng as of 1.0pre5.

```
<Package name='glibc' type='rpm'>
    <Instance simplefile='glibc-2.3.4-2.25.x86_64.rpm' version='2.3.4' release='2.25' arch=
        <Ignore name='/etc/rpc'/>
    </Instance>
    <Instance simplefile='glibc-2.3.4-2.25.i686.rpm' version='2.3.4' release='2.25' arch='i
</Package>
```

pkg_checks If pkg_checks = false the version information is not required. If pkg_checks = true the full information is needed as normal.

For YUMng a minimal entry is

```
<Package name="bcfg2" type="yum" pkg_checks="False"/>
```

In fact for YUMng, with pkg_checks = false, any combination of the nevra attributes that will build a valid yum package name (see the Misc heading on the yum man page) is valid.

```
<Package name="bcfg2" type="yum" pkg_checks="False" arch="x86_64"/>
```

For RPMng a minimal entry is

```
<Package name="bcfg2" type="rpm" pkg_checks="False" simplefile="bcfg2-0.9.4-0.0pre1.noarch
```

verify_fail_action The way I have Bcfg2 configured for my development systems. This way it reports bad, but doesn't do anything about it.

```
<Package name='bcfg2' type='rpm'>
    <Instance simplefile='bcfg2-0.9.3-0.0pre5.noarch.rpm' version='0.9.3' release='0.0pre5'
</Package>
```

6.2 Other client-related documentation

6.2.1 Agent Functionality using SSH

The Bcfg2 agent code provides the ability to trigger a client update from the server using a secure mechanism that is restricted to running the Bcfg2 client with the options the agent was started with. This same capability is provided by SSH keypairs, if properly configured. Setup is pretty easy:

1. Create an ssh keypair that is to be used solely for triggering Bcfg2 client runs. This key may or may not have a password associated with it; a keyphrase will make things more secure, but will require a person to enter the key passphrase, so it will not be usable automatically.:

   ```
   $ ssh-keygen -t dsa -b 1024 -f /path/to/key -N ""
   Generating public/private dsa key pair.
   Your identification has been saved in /path/to/key.
   Your public key has been saved in /path/to/key.pub.
   The key fingerprint is:
   aa:25:9b:a7:10:60:f3:eb:2b:ae:4b:1a:42:1b:63:5d desai@ubik
   ```

2. Add this this public key to root's authorized_keys file, with several commands prepended to it:

   ```
   command="/usr/sbin/bcfg2 -q <other options>",no-port-forwarding,no-X11-forwarding,no
   ```

   This key is now only useful to call the Bcfg2 client, from the Bcfg2 server's ip address. If PermitRootLogin was set to no in sshd_config, you will need to set it to forced-commands-only. Adding a & to the end of the command will cause the command to immediately return.

3. Now, to cause a client to reconfigure, call:

   ```
   $ ssh -i /path/to/key root@client /usr/sbin/bcfg2
   ```

   Note that you will not be able to alter the command line options from the ones specified in authorized_keys in any way. Also, it is not needed that the invocation of Bcfg2 in the ssh command match. The following will have the same result.:

```
$ ssh -i /path/to/key root@client /bin/true
```

If a passphrase was used to create the keypair, then it will need to be entered here.

See Also

SSH "triggers" (from Ganneff's Little Blog)

6.2.2 Client Debugging

When working on the Bcfg2 client, it is helpful to employ a few specific techniques to isolate and remedy problems.

First, running the client with the -f flag allows configuration from a local file, rather than querying the server. This helps rule out server configuration problems, and allows for rapid development. For example: `bcfg2 -f test-config.conf` with the following test-config.conf:

```
<Configuration>
    <Bundle name="ssh-tests">
        <Service type="launchd" name="com.openssh.sshd" status="on" />
    </Bundle>
</Configuration>
```

Next, it is important to look at the interactive mode. This is similar to the interactive mode on the server and provides an interactive Python interpreter with which one may manipulate all the objects in the client. It will setup all the infrastructure so you will have the appropriate objects to play with. It will run the client through once, then present you with an interpreter. Try it out with: `python -i /usr/bin/bcfg2` or, for more fun, a local config file and also enable Debugging and Verbose output with *-d* and *-v*, yielding `python -i /usr/bin/bcfg2 -d -v -f test-config.conf`.

Now we just explore; use `dir()` to examine different objects in the client, or run a reconfiguration again by calling *client.run()*

6.2.3 Client Metadata

This page describes ClientMetadata objects. These are used to describe clients in terms of a variety of parameters, group memberships, and so forth.

Construction

ClientMetadata instances are constructed whenever the server needs to recognize a client. This occurs in every aspect of client server interaction:

- Probing
- Configuration Generation
- Statistics Upload

This construction process spans several server plugins. The *Metadata* is responsible for initial instance creation, including the client hostname, profile, and basic group memberships. After this initial creation, Connector plugins (such as *Probes* or *Properties*) can add additional group memberships for clients. These memberships are merged into the instance; that is, the new group memberships are treated as if they were included in groups.xml. If any of these groups are defined in groups.xml, then groups included there are included in the ClientMetadata instance group list. At the end of this process, the ClientMetadata instance has its complete set of group memberships. At this point, each connector plugin has the opportunity to return an additional object which will be placed in an attribute corresponding to the Connector name. For example, the Probes plugin returns a dictionary of probe name to probe result mappings for the client. This dictionary is available as the "Probes" attribute. With this, ClientMetadata resolution is complete, and the ClientMetadata instance can be used by the rest of the system.

Contents

ClientMetadata instances contain all of the information needed to differentiate clients from one another. This data includes:

- hostname
- groups
- profile group
- address information (if specified)

ClientMetadata instances also contain a query object. This can be used to query the metadata of other clients. Currently, several methods are supported. In this table, we refer to the instance as meta. Each of these is a function that must be called.

Name	Description	Return Type
meta.query.names_by_groups([group list])	Returns names of clients which are members of all groups	List of client names
meta.query.names_by_profile(profile)	Returns names of clients which use profile group	List of client names
meta.query.all_clients()	Returns names of all clients	List of client names
meta.query.all_groups()	Returns names of all groups	List of group names
meta.query.all()	Returns metadata for all clients	List of ClientMetadata instances
meta.query.by_name(name)	Returns metadata for named client	ClientMetadata instance
meta.query.by_groups([group list])	Returns metadata for all members of all groups	List of ClientMetadata instances
meta.query.by_profile(profile)	Returns metadata for all profile havers	List of ClientMetadata instances

In general, there is no substantial benefit to using name returning versions of the query functions; metadata resolution is (in general) fast.

6.2.4 Client modes

Dryrun mode

Dryrun mode (-n) prevents the client from making changes, but gives you some insight into the state of the machine. This mode is also useful if you simply want to gather data from the client into the reporting system.

Interactive mode

The client can be run interactively (-I) so that you are able to step through each operation in order to see what the client is doing.

Paranoid mode

Paranoid mode creates a backup of a local configuration file before Bcfg2 replaces the file. This allows for easier recovery by the local administrator.

How do I use it?

1. In the Bcfg2 repository, put *paranoid='true'* in the `info.xml` file.
2. On the client, create `/var/cache/bcfg2` (or specify an alternate path in the [paranoid] section of `/etc/bcfg2`).
3. On the client, run *bcfg2* with the *-P* option (alternatively, you can set *paranoid* to *true* in the **[client]** section of `bcfg2.conf`).

This will save a copy of the replaced file in `/var/cache/bcfg2`, but it'll be named as the path to the file with /'s replaced by _'s. For example, the old `/etc/hosts` will be named `/var/cache/bcfg2/etc_hosts`.

Extra configuration

New in version 1.0.0. Here is an example of how to use some of the extra paranoid features available. For the following section in `bcfg2.conf` (client-side):

```
[paranoid]
path = /my/custom/backup/path
max_copies = 5
```

You will have the file backups store in `/my/custom/backup/path`. This will also keep the five most recent backups of files.

Altering the global metadata to enable paranoid mode for all files

You may also want to just globally enable the *paranoid* attribute for all files distributed to clients from your Bcfg2 server. You can accomplish this by adding a global metadata override in your `bcfg2.conf` (server-side) with the following syntax:

```
[mdata]
paranoid=true
```

This will override the default value of "paranoid=false" and change it to "true" which will cause every file you add or update in your Bcfg2 repo to backup on the client as specified in your client config.

Overall client service mode

New in version 1.0.0. Overall client service mode. Specified on the client using `-s <service mode>`.

- default
  - perform all service manipulations
- disabled
  - perform no service manipulations
- build
  - attempt to stop all services started
  - deprecates/replaces -B

CHAPTER

SEVEN

THE BCFG2 REPORTING SYSTEM

Bcfg2's reporting system is its killer feature. It allows administrators to gain a broad understanding of the configuration state of their entire environment. It summarizes

- Configuration changes and when they were made
- Discrepancies between the specification and current client states
  - Clients can be grouped by misconfiguration type
- Configuration entries that are not specified
- Overall client summaries according to these types

There are two systems, the old system, which builds static reports based on a series of XSLT stylesheets and a new dynamic reporting system that uses django and a database backend.

7.1 Bcfg2 Static Reporting System

The Bcfg2 reporting system collects and displays information about the operation of the Bcfg2 client, and the configuration states of target machines.

7.1.1 Goals

The reporting system provides an interface to administrators describing a few important tasks

- Client configuration state, particularly aspects that do not match the configuration specification. Information about bad and extra configuration elements is included.
- Client execution results (a list of configuration elements that were modified)
- Client execution performance data (including operation retry counts, and timings for several critical execution regions)

This data can be used to understand the current configuration state of the entire network, the operations performed by the client, how the configuration changes propagate, and any reconfiguration operations that have failed.

7.1.2 Retention Model

The current reporting system stores statistics in an XML data store, by default to {{{<repo>/etc/statistics.xml}}}. It retains either one or two statistic sets per host. If the client has a clean configuration state, the most recent (clean) record is retained. If the client has a dirty configuration state, two records are retained. One record is the last clean record. The other record is the most recent record collected, detailing the incorrect state.

This retention model, while non-optimal, does manage to persistently record most of the data that users would like.

7.1.3 Setup

In order to configure your Bcfg2 server for receiving reports, you will need to list the Statistics plugin in the plugins line of your `bcfg2.conf`. You will also need a [statistics] section in your `bcfg2.conf`. You can find out more about what goes there in the `bcfg2.conf` manpage.

7.1.4 Output

Several output reports can be generated from the statistics store with the command line tool {{{bcfg2-build-reports}}}.

- Nodes Digest
- Nodes Individual
- Overview Statistics
- Performance

The data generated by these reports can be delivered by several mechanisms:

- HTML
- Email
- RSS

7.1.5 Shortcomings and Planned Enhancements

When designing the current reporting system, we were overly concerned with the potential explosion in data size over time. In order to address this, we opted to use the retention scheme described above. This approach has several shortcomings:

- A comprehensive list of reconfiguration operations (with associated timestamps) isn't retained
- Client results for any given day (except the last one) aren't uniformly retained. This means that inter-client analysis is difficult, if not impossible

We plan to move to a database backend to address the dataset size problem and start retaining all information. The move to a SQL backend will allow many more types of queries to be efficiently processed. It will also make on-demand reports simpler.

Other sorts of information would also be useful to track. We plan to add the ability to tag a particular configuration element as security related, and include this in reports. This will aid in the effective prioritization of manual and failed reconfiguration tasks.

Capability Goals (posed as questions)

- What machines have not yet applied critical updates?
- How long did critical updates take to be applied?
- What configuration did machine X have on a particular date?
- When did machine X perform configuration update Y?

7.2 Bcfg2 Dynamic Reporting System

New in version 0.8.2.

7.2.1 Installation

Prerequisites

- sqlite3
- pysqlite2
- Django
- mod-python

Install

Be sure to include the specified fields included in the example `bcfg2.conf` file. These can be specified in either `/etc/bcfg2.conf`, if it is readable by the webserver user, or `/etc/bcfg2-web.conf`. Any database supported by Django can be used. If you are not using sqlite (the default choice), please see the *Notes on Alternative Databases* section below.

Note: Distributed environments can share a single remote database for reporting.

The recommended statistics plugin is DBStats. To use it, add it to the **plugins** line in your `bcfg2.conf`. Alternatively, the Statistics plugin can be used in conjunction with a crontab entry to run `/usr/sbin/bcfg2-admin reports load_stats`.

Detailed installation instructions can be found *here*.

Apache configuration for web-based reports

Note: Reports no longer needs to be installed at the root URL for a given host. Therefore, reports no longer require their own virtual host.

In order to make this work, you will need to specify your web prefix by adding a **web_prefix** setting in the [statistics] section of your `bcfg2.conf`.

An example site config is included below for the vhost "reports.mcs.anl.gov":

```
<VirtualHost reports.mcs.anl.gov>
        ServerAdmin webmaster@mcs.anl.gov
        ServerName reports.mcs.anl.gov
        DocumentRoot /var/www/reports
        <Directory /var/www/reports>
                Order deny,allow
                Deny from all
                Allow from localhost #you may want to change this
                AllowOverride None
        </Directory>

        # Possible values include: debug, info, notice, warn, error, crit,
        # alert, emerg.
        LogLevel warn

        ServerSignature Off

        # Stop TRACE/TRACK vulnerability
        <IfModule mod_rewrite.c>
                RewriteEngine on
                RewriteCond %{REQUEST_METHOD} ^(TRACE|TRACK)
                RewriteRule .* - [F]
        </IfModule>
<Location "/">
        SetHandler python-program
        PythonHandler django.core.handlers.modpython
        SetEnv DJANGO_SETTINGS_MODULE Bcfg2.Server.Reports.settings
        PythonDebug On
</Location>
<Location "/site_media/">
        SetHandler None
</Location>
</VirtualHost>
```

The first three lines of this configuration must be customized per site.

The `bcfg2-tarball/reports/site_media/` directory needs to be copied to `/var/www/reports/site_media/` It could live anywhere; as long as the contents are accessible on the virtual host at `/site_media/`.

At this point you should be able to point your web browser to the virtualhost you created and see the new reports

Example WSGI configuration

entry.wsgi:

```
import os, sys
os.environ['DJANGO_SETTINGS_MODULE'] = 'Bcfg2.Server.Reports.settings'
import django.core.handlers.wsgi
application = django.core.handlers.wsgi.WSGIHandler()
```

Apache conf:

```
Alias /bcfg2reports/site_media "/path/to/site_media"
<Directory /path/to>
    Order deny,allow
    Allow from all
    AllowOverride None
</Directory>
# If Python is installed in a non-standard prefix:
#WSGIPythonHome /python/prefix
#WSGIPythonPath /python/prefix/lib/python2.6/site-packages
WSGIScriptAlias /bcfg2reports "/another/path/to/entry.wsgi"
```

Notes on Alternative Databases

If you choose to use a different database, you'll need to edit `/etc/bcfg2.conf`. These fields should be updated in the [statistics] section:

- database_engine
  - ex: database_engine = mysql
  - ex: database_engine = postgresql_psycopg2
- database_name
- database_user
- database_password
- database_host
- database_port (optional)

7.2.2 Summary and Features

The new reporting system was implemented to address a number of deficiencies in the previous system. By storing statistics data in a relational database, we are now able to view and analyze more information about the state of the configuration, including information about previous configuration. Specific features in the new system include:

- The ability to look at a *Calendar Summary* with past statistics information.
- More recent data concerning hosts.

- Additional information display in reports. Primarily, reasons for *configuration item verification failure* are now accessible.
- Instead of static pages, pages are generated on the fly, allowing users to drill down to find out about a *specific host*, rather than only having one huge page with too much information.

7.2.3 Planned improvements include

- Accounts, customized displays for each admin. And privacy of config data.
- Config browsing capabilities; to look at your config in an interesting way.
- Fixing all the known bugs from below.

Unfortunately with all the improvements, there are a few less exciting elements about the new reporting system. The new reporting system moves away from static pages and towards a real web application, which causes mainly problems with dependencies and makes installation more difficult. This should become less of a problem over time as we develop a better installation process for a web application.

7.2.4 Usage

bcfg2-admin reports (command line script)

The bcfg2-admin tool provides management and maintenance capabilities for the reporting database. A few useful Django commands are provided as well.

- init: Initialize a new database
- load_stats: Load statistics data from the Statistics plugin into the database. This was importscript.py.
- scrub: Scrub the database for duplicate reasons.
- update: Apply any updates to the reporting database. Unlike the syncdb command, this will modify existing tables.

Django commands

- syncdb: Create the tables for any models not installed. Django will not modify any existing tables.
- sqlall: Print the sql statements used to create the database. Note: This does not show the fixture data.
- validate: Validate the database against the current models.

bcfg2-reports (command line script)

bcfg2-reports allows you to retrieve data from the database about clients, and the states of their current interactions. It also allows you to change the expired/unexpired states.

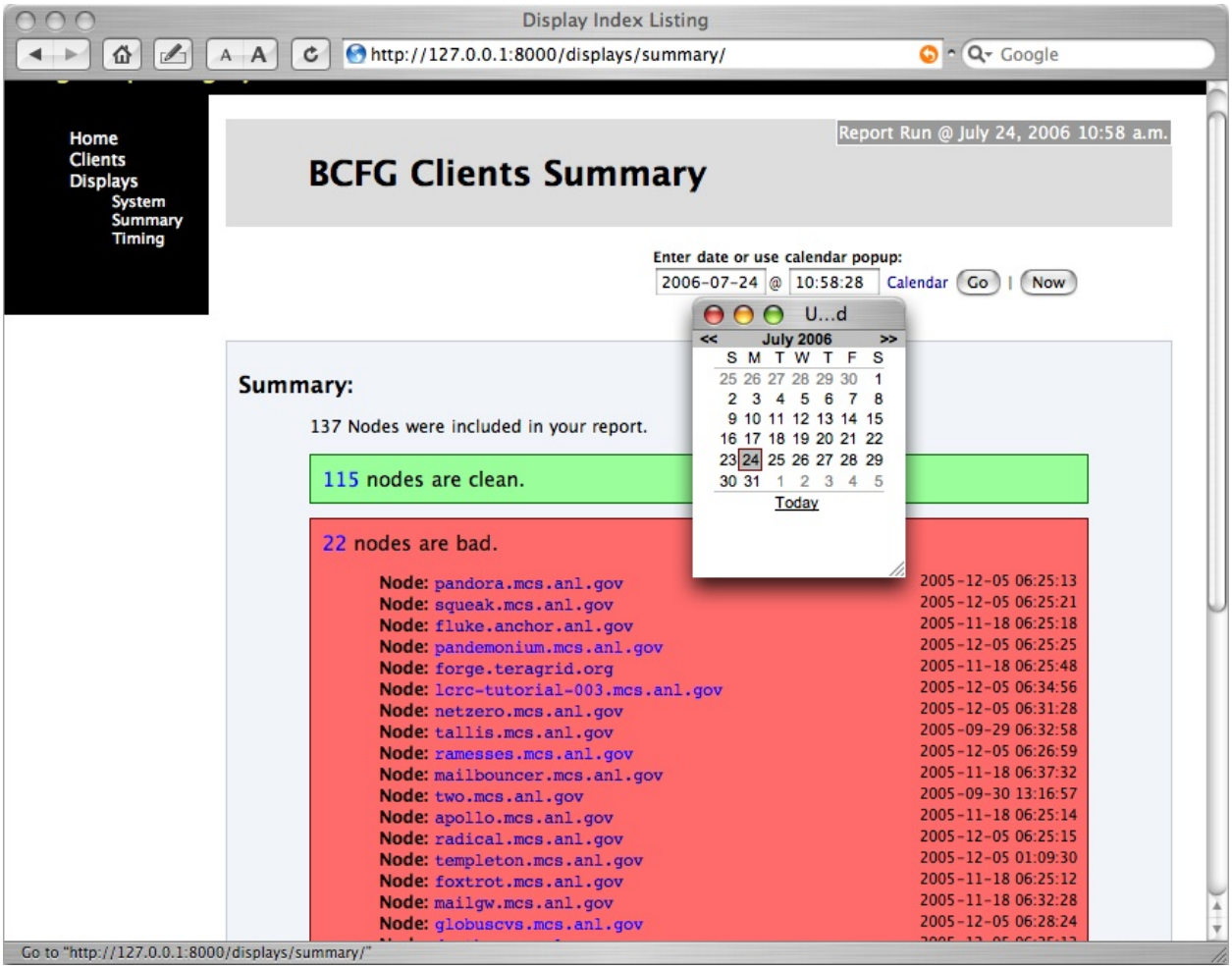
The utility runs as a standalone application. It does, however, use the models from `/src/lib/Server/Reports/reports/models.py`.

A number of different options can be used to change what bcfg2-reports displays:

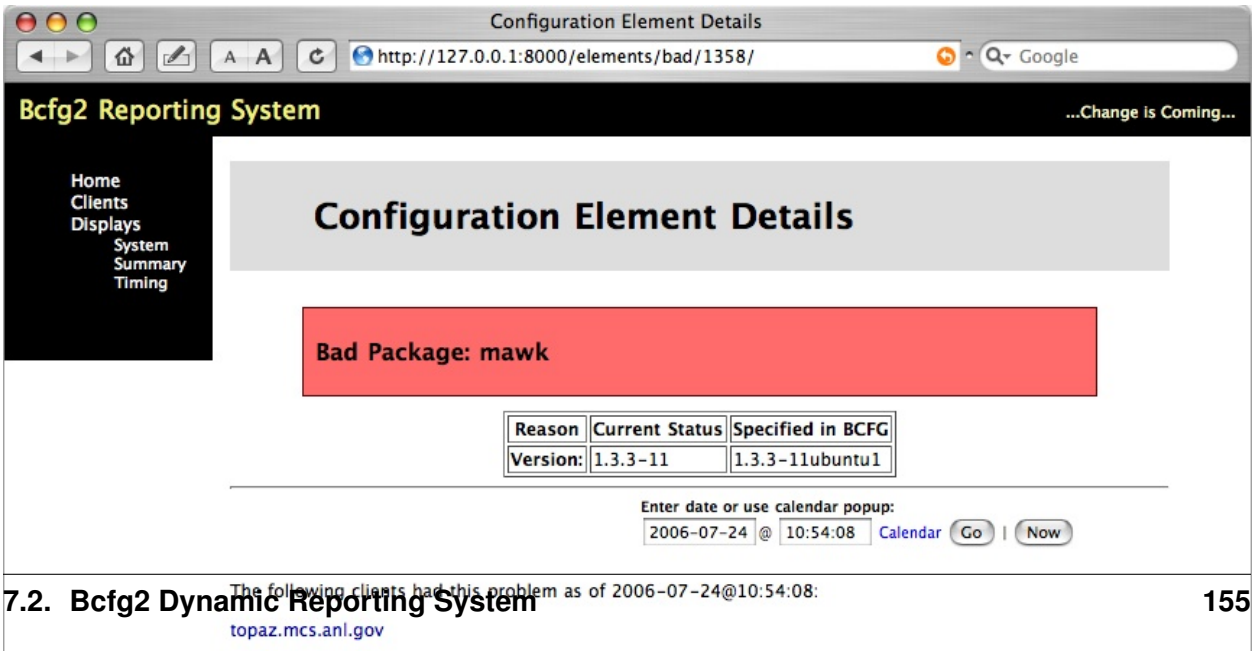
```
Usage: python bcfg2-reports [option] ...

Options and arguments (and corresponding environment variables):
-a                      : shows all hosts, including expired hosts
-b NAME                 : single-host mode - shows bad entries from the
                          current interaction of NAME
-c                      : shows only clean hosts
-d                      : shows only dirty hosts
-e NAME                 : single-host mode - shows extra entries from the
                          current interaction of NAME
-h                      : shows help and usage info about bcfg2-reports
-s NAME                 : single-host mode - shows bad and extra entries from
                          the current interaction of NAME
-x NAME                 : toggles expired/unexpired state of NAME
--badentry=KIND,NAME    : shows only hosts whose current interaction has bad
                          entries in of KIND kind and NAME name; if a single
                          argument ARG1 is given, then KIND,NAME pairs will be
                          read from a file of name ARG1
--extraentry=KIND,NAME  : shows only hosts whose current interaction has extra
                          entries in of KIND kind and NAME name; if a single
                          argument ARG1 is given, then KIND,NAME pairs will be
                          read from a file of name ARG1
--fields=ARG1,ARG2,...  : only displays the fields ARG1,ARG2,...
                          (name,time,state)
--sort=ARG1,ARG2,...    : sorts output on ARG1,ARG2,... (name,time,state)
--stale                 : shows hosts which haven't run in the last 24 hours
```

7.2.5 Screenshots

Calendar Summary



Item detail



CHAPTER

EIGHT

BCFG2 DEVELOPMENT

There are several ways users can contribute to the Bcfg2 project.

- Developing code
- Testing prereleases
- Reporting bugs
- Adding to the common repository
- Improving the wiki and writing documentation

This section will outline some things that can help you get familiar with the various areas of the Bcfg2 code.

Send patches to the *mailinglist* or create a trac ticket with the patch included. In order to submit a ticket via the trac system, you will need to create a session by clicking on the Preferences link and filling out/saving changes to the form. In order to be considered for mainline inclusion, patches need to be BSD licensed. The most convenient way to prepare patches is by using `git diff` inside of a source tree checked out of git.

The source tree can be checked out by running:

```
git clone git://git.mcs.anl.gov/bcfg2.git
```

Users wishing to contribute on a regular basis can apply for direct git access. Mail the *mailinglist* for details.

8.1 Tips for Bcfg2 Development

1. Focus on either the client or server code. This focuses the development process down to the precise pieces of code that matter for the task at hand.
   - If you are developing a client driver, then write up a small configuration specification that includes the needed characteristics.
   - If you are working on the server, run `bcfg2-info` and use to assess the code.
2. Use the python interpreter. One of python's most appealing features is interactive use of the interpreter.
   - If you are developing for the client-side, run `python -i /usr/sbin/bcfg2` with the appropriate bcfg2 options. This will cause the python interpreter to continue running, leaving all variables intact. This can be used to examine data state in a convenient fashion.

- If you are developing for the server side, use `bcfg2-info` and the "debug" option. This will leave you at a python interpreter prompt, with the server core loaded in the variable "bcore".

3. Use `pylint` obsessively. It raises a lot of style-related warnings which can be ignored, but most all of the errors are legitimate.
4. If you are doing anything with Regular Expressions, Kodos and re-try are your friends.

8.2 Environment setup for development

- Check out a copy of the code:

```
svn co https://svn.mcs.anl.gov/repos/bcfg/trunk/bcfg2
```

- Create link to `src/lib`:

```
cd bcfg2
ln -s src/lib Bcfg2
```

- Add `bcfg2/src/sbin` to your PATH environment variable
- Add `bcfg2` to your PYTHONPATH environment variable

8.3 Writing A Client Tool Driver

This page describes the step-by-step process of writing a client tool driver for a configuration element type. The included example describes an existing driver, and the process that was used to create it.

1. Pick a name for the driver. In this case, we picked the name RPM.
2. Add "RPM" to the `__all__ list` in `src/lib/Client/Tools/__init__.py`
3. Create a file in `src/lib/Client/Tools` with the same name (RPM.py)
4. Create a class in this file with the same name (`class RPM`)
   - If it handles `Package` entries, subclass `Bcfg2.Client.Tools.PkgTool` (from here referenced as branch [P])
   - If it handles `Service` entries, subclass `Bcfg2.Client.Tools.SvcTool` (from here referenced as branch [S])
   - Otherwise, `subclass Bcfg2.Client.Tools.Tool` (from here referenced as branch [T])
5. Set `__name__` to "RPM"
6. Add any required executable programs to `__execs__`
7. Set `__handles__` to a list of `(entry.tag, entry.get('type'))` tuples. This determines which entries the Tool module can be used on. In this case, we set `__handles__ = [('Package', 'rpm')]`.

8. Add verification. This method should return True/False depending on current entry installation status.
    - [T] Add a Verify<entry.tag> method.
    - [P] Add a VerifyPackage method.
    - [S] Add a VerifyService method.
    - In the failure path, the current state of failing entry attributes should be set in the entry, to aid in auditing. (For example, if a file should be mode 644, and is currently mode 600, then set attribute current_perms='600' in the input entry)
9. Add installation support. This method should return True/False depending on the results of the installation process.
    - [T,S] Add an Install<entry.tag> method.
    - [P] The PkgTool baseclass has a generic mechanism for performing all-at-once installations, followed, in the case of failures, by single installations. To enable this support, set the pkgtype attribute to the package type handled by this driver. Set the `pkgtool` to a tuple ("command string %s", ("per-package string format", [list of package entry fields])). For RPM, we have `setup pkgtool = ("rpm --oldpackage --replacepkgs --quiet -U %s", ("%s", ["url"]))`
10. Implement entry removal
    - [T,S] Implement a `Remove` method that removes all specified entries (`prototype Remove(self, entries)`)
    - [P] Implement a `RemovePackages` that removes all specified entries (same prototype as Remove)
11. Add a `FindExtra` method that locates entries not included in the configuration. This may or may not be required, certain drivers do not have the capability to find extra entries.
12. [P] Package drivers require a `RefreshPackages` method that updates the internal representation of the package database.

8.3.1 Writing Tool Driver Methods

1. Programs can be run using `self.cmd.run`. This function returns a (return code, stdout list) tuple.
2. The configuration is available as `self.config`
3. Runtime options are available in a dictionary as `self.setup`
4. Informational, error, and debug messages can be produced by running `self.logger.info/error/debug`.

8.4 Bcfg2 Plugin development

While the Bcfg2 server provides a good interface for representing general system configurations, its plugin interface offers the ability to implement configuration interfaces and representation tailored to problems

encountered by a particular site. This chapter describes what plugins are good for, what they can do, and how to implement them.

8.4.1 Bcfg2 Plugins

Bcfg2 plugins are loadable python modules that the Bcfg2 server loads at initialization time. These plugins can contribute to the functions already offered by the Bcfg2 server or can extend its functionality. In general, plugins will provide some portion of the configuration for clients, with a data representation that is tuned for a set of common tasks. Much of the core functionality of Bcfg2 is implemented by several plugins, however, they are not special in any way; new plugins could easily supplant one or all of them.

The following table describes the various functions of bcfg2 plugins.

Name	Description
Probes	Plugins can issue commands to collect client-side state (like hardware inventory) to include in client configurations
ConfigurationEntry List	Plugins can construct a list of per-client configuration entry lists to include in client configurations.
ConfigurationEntry contents	Literal values for configuration entries
XML-RPC functions	Plugins can export function calls that expose internal functions.

8.5 Writing Bcfg2 Plugins

Bcfg2 plugins are python classes that subclass from Bcfg2.Server.Plugin.Plugin. Several plugin-specific values must be set in the new plugin. These values dictate how the new plugin will behave with respect to the above four functions. The following table describes all important member fields.

Name	Description	Format
__name__	The name of the plugin	string
__version__	The plugin version (generally tied to revctl keyword expansion)	string
__author__	The plugin author.	string
__author__	The plugin author.	string
__rmi__	Set of functions to be exposed as XML-RPC functions	List of function names (strings)
Entries	Multidimentional dictionary of keys that point to the function used to bind literal contents for a given configuration entity.	Dictionary of ConfigurationEntityType, Name keys, and function reference values
BuildStructures	Function that returns a list of the structures for a given client	Member function
GetProbes	Function that returns a list of probes that a given client should execute	Member function
ReceiveData	Function that accepts the probe results for a given client.	Member function

8.5.1 Example Plugin

```
import Bcfg2.Server.Plugin
class MyPlugin(Bcfg2.Server.Plugin.Plugin):
    '''An example plugin'''
    # All plugins need to subclass Bcfg2.Server.Plugin.Plugin
    __name__ = 'MyPlugin'
    __version__ = '1'
    __author__ = 'me@me.com'
    __rmi__ = ['myfunction']
    # myfunction is not available remotely as MyPlugin.myfunction

    def __init__(self, core, datastore):
        Bcfg2.Server.Plugin.Plugin.__init__(self, core, datastore)
        self.Entries = {'Path':{'/etc/foo.conf': self.buildFoo}}

    def myfunction(self):
        '''function for xmlrpc rmi call'''
        #do something
        return True

    def buildFoo(self, entry, metadata):
        '''Bind per-client information into entry based on metadata'''
        entry.attrib.update({'type':'file', 'owner':'root', 'group':'root', 'perms':'644'})
        entry.text = '''contents of foo.conf'''
```

8.5.2 Example Connector

```
import Bcfg2.Server.Plugin

class Foo(Bcfg2.Server.Plugin.Plugin,
          Bcfg2.Server.Plugin.Connector):
    '''The Foo plugin is here to illustrate a barebones connector'''
    name = 'Foo'
    version = '$Revision: $'
    experimental = True

    def __init__(self, core, datastore):
        Bcfg2.Server.Plugin.Plugin.__init__(self, core, datastore)
        Bcfg2.Server.Plugin.Connector.__init__(self)
        self.store = XMLFileBacked(self.data, core.fam)

    def get_additional_data(self, metadata):

        mydata = {}
        for data in self.store.entries['foo.xml'].data.get("foo", []):

            mydata[data] = "bar"

        return  dict([('mydata', mydata)])

    def get_additional_groups(self, meta):
        return self.cgroups.get(meta.hostname, list())
```

8.6 Server Plugin Types

8.6.1 Generator

Generator plugins contribute to literal client configurations

8.6.2 Structure

Structure Plugins contribute to abstract client configurations

8.6.3 Metadata

Signal metadata capabilities

8.6.4 Connector

Connector Plugins augment client metadata instances

8.6.5 Probing

Signal probe capability

8.6.6 Statistics

Signal statistics handling capability

8.6.7 Decision

Signal decision handling capability

8.6.8 Version

Interact with various version control systems

8.7 Writing Bcfg2 Specification

Bcfg2 specifications are logically divided in to three areas:

- Metadata
- Abstract
- Literal

The metadata portion of the configuration assigns a client to its profile group and to its non-profile groups. The profile group is assigned in `Metadata/clients.xml` and the non profile group assignments are in `Metadata/groups.xml`.

The group memberships contained in the metadata are then used to constuct an abstract configuration for the client. An abstract configuration for a client identifies the configuration entities (packages, configuration files, service, etc) that a client requires, but it does not identify them explicitly. For instance an abstract configuration may identify that a client needs the Bcfg2 package with

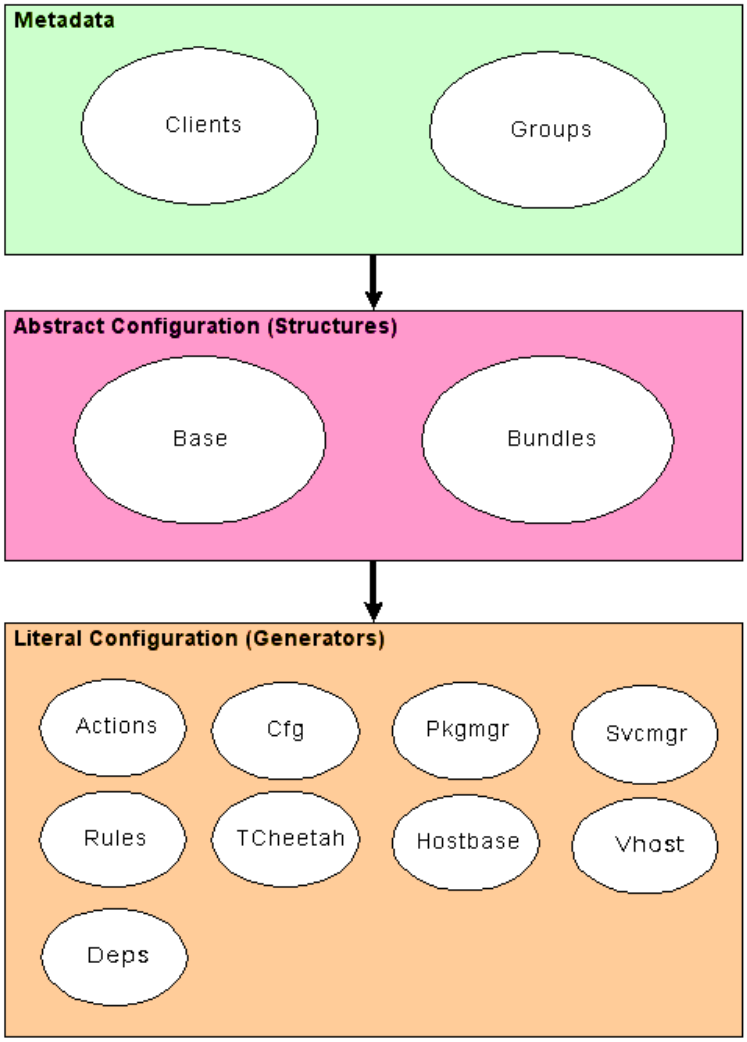
```
<Package name=bcfg2/>
```

but this does not explicitly identify that an RPM package version 0.9.2 should be loaded from http://rpm.repo.server/bcfg2-1.0.1-0.1.rpm. The abstract configuration is defined in the xml configuration files for the Base and Bundles plugins.

A combination of a clients metadata (group memberships) and abstract configuration is then used to generate the clients literal configuration. For instance the above abstract configuration entry may generate a literal configuration of

```
<Package name='bcfg2' version='1.0.1-0.1' type='yum'/>
```

A clients literal configuration is generated by a number of plugins that handle the different configuration entities.


Metadata
Clients
Groups
Abstract Configuration (Structures)
Base
Bundles
Literal Configuration (Generators)
Actions
Cfg
Pkgmgr
Svcmgr
Rules
TCheetah
Hostbase
Vhost
Deps

8.8 Writing Server Plugins

8.8.1 Metadata

If you would like to define your own Metadata plugin (to extend/change functionality of the existing Metadata plugin), here are the steps to do so. We will call our new plugin *MyMetadata*.

1. Add MyMetadata.py

```
__revision__ = '$Revision$'

import Bcfg2.Server.Plugins.Metadata

class MyMetadata(Bcfg2.Server.Plugins.Metadata.Metadata):
    '''This class contains data for bcfg2 server metadata'''
    __version__ = '$Id$'
    __author__ = 'bcfg-dev@mcs.anl.gov'

def __init__(self, core, datastore, watch_clients=True):
    Bcfg2.Server.Plugins.Metadata.Metadata.__init__(self, core, datastore, watch_cli
```

2. Add MyMetadata to `src/lib/Server/Plugins/__init__.py`
3. Replace Metadata with MyMetadata in the plugins line of bcfg2.conf

8.9 Packages

In order to support a given client package tool driver, that driver must support use of the auto value for the version attribute in Package entries. In this case, the tool driver views the current state of available packages, and uses the underlying package manager's choice of correct package version in lieu of an explicit, centrally-specified, version. This support enables Packages to provide a list of Package entries with version='auto'. Currently, the APT and YUMng drivers support this feature. Note that package management systems without any network support cannot operate in this fashion, so RPMng and SYSV will never be able to use Packages. Emerge, Zypper, IPS, and Blastwave all have the needed features to be supported by Packages, but support has not yet been written.

Packages fills two major functions in configuration generation. The first is to provide entry level binding support for Package entries included in client configurations. This function is quite easy to implement; Packages determines (based on client group membership) if the package is available for the client system, and which type it has. Because version='auto' is used, no version determination needs to be done.

The second major function is more complex. Packages ensures that client configurations include all package-level prerequisites for package entries explicitly included in the configuration. In order to support this, Packages needs to directly process network data for package management systems (the network sources for apt or yum, for examples), process these files, and build data structures describing prerequisites and the providers of those functions/paths. To simplify implementations of this, there is a generic base class (Bcfg2.Server.Plugins.Packages.Source) that provides a framework for fetching network data via HTTP, processing those sources (with subclass defined methods for processing the specific format provided

by the tool), a generic dependency resolution method, and a caching mechanism that greatly speeds up server/bcfg2-info startup.

Each source type must define:

- a get_urls attribute (and associated urls property) that describes the URLS where to get data from.
- a read_files method that reads and processes the downloaded files

Sources may define a get_provides method, if provides are complex. For example, provides in rpm can be either rpm names or file paths, so multiple data sources need to be multiplexed.

The APT source in `src/lib/Server/Plugins/Packages.py` provides a relatively simple implementation of a source.

8.10 Testing

8.10.1 Testing Prereleases

Before each release, several prereleases will be tagged. It is helpful to have users test these releases (when feasible) because it is hard to replicate the full range of potential reconfiguration situations; between different operating systems, system management tools, and configuration specification variation, there can be large differences between sites.

For more details please visit Tracking Development Releases of Bcfg2 .

8.10.2 Upgrade Testing

This section describes upgrade procedures to completely test the client and server. These procedures can be used for either pre-release testing, or for confidence building in a new release.

Server Testing

1. Ensure that the server produces the same configurations for clients

- Before the upgrade, generate all client configurations using the buildall subcommand of bcfg2-info. This subcommand takes a directory argument; it will generate one client configuration in each file, naming each according to the client name.

```
mgt1:~/bcfg# bcfg2-info
Filesystem check 1 of 25
...
> buildall /path/to/cf-old
Generated config for fs2.bgl.mcs.anl.gov in 1.97310400009 seconds
Generated config for fs13.bgl.mcs.anl.gov in 1.47958016396 seconds
...
```

Take notice of any messages produced during configuration generation. These generally reflect minor issues in the configuration specification. Ideally, they should be fixed.

- Upgrade the server software
- Generate all client configurations in a second location using the new software. Any tracebacks reflect bugs, and should be filed in the ticketing system. Any new messages should be carefully examined.
- Compare each file in the old directory to those in the new directory using `bcfg2-admin compare -r /old/directory /new/directory`

```
mgt1:~/bcfg# bcfg2-admin compare -r cf-old/ cf-new/
Entry: fs2.bgl.mcs.anl.gov.xml
Entry: fs2.bgl.mcs.anl.gov.xml good
Entry: fs13.bgl.mcs.anl.gov.xml
Entry: fs13.bgl.mcs.anl.gov.xml good
Entry: login1.bgl.mcs.anl.gov.xml
 ConfigFile /bin/whatami contents differ
 ConfigFile /bin/whatami differs (in bundle softenv)
Entry: login1.bgl.mcs.anl.gov.xml bad
```

  This can be used to compare configurations for single clients, or different clients.

2. Compare old and new group diagrams (using `bcfg2-admin viz`)

Client Testing

Run the client in dry-run and non-dry-run mode; ensure that multiple runs produce consistent results.

8.11 Documentation

There are two parts of documentation in the Bcfg2 project:

- The wiki
- The manual

8.11.1 The wiki

A python-based Trac instance is used for the Bcfg2 website. The Wiki part of the website can be edited after you have successful logged in. For the login is a vaild OpenID provider needed and an interaction from an administrator. Please request your access to the Wiki on the *mailinglist* or in the *IRC Channel*.

8.11.2 The manual

The source for the manual is located in the *doc/* directory in the SVN repository or in the source tarball. All files are written in rst (ReStructuredText). For the build process we are using Sphinx.

Building the Manual

- Install the prerequisites. Docutils and Sphinx are needed to build.
  - For Debian (Lenny) the tools are available in the backports repository; installation can be done with the following:

    ```
    apt-get -t lenny-backports install python-sphinx
    ```

  - The needed tools for Fedora based systems are in the Fedora Package Collection; installation can be done easily with Yum:

    ```
    yum -y install python-sphinx python-docutils
    ```

  - Additionally, to build the PDF version:
  - LaTeX
  - pdftex
- Download the source. Please refer to *Download* for more details.
- Building the HTML version, run the following command in the *doc/* directory. The output will appear in *../build/sphinx/html*:

  ```
  python setup.py build_sphinx
  ```

- Building the PDF version

  ```
  python setup.py build_sphinx --builder=latex
  cd build/sphinx/latex
  make
  ```

The latest version of the manual

The latest version of the manual can always be found on the Four Kitchens server.

This is an auto-updated from the Launchpad mirror.

8.12 Documentation Style Guide for Bcfg2

This is a style guide to use when creating documentation for Bcfg2. It is meant to be helpful, not a hinderence.

8.12.1 Basics

Bcfg2

When referring to project, Bcfg2 is the preferred use of cases.

Monospace fonts

When referring to commands written on the command line use monospace fonts.

Repository

When used alone this refers to a Bcfg2 *repository*. When there is a chance for confusion, for instance in documents also talking about *VCS*, be sure to use the longer Bcfg2 *repository*.

8.13 Emacs + YASnippet mode

This page describes using emacs with YASnippet mode with a set of snippets that allow quick composition of bundles and base files. More snippets are under development.

1. Download YASnippet from http://code.google.com/p/yasnippet/
2. Install it into your emacs load path (typically ~/.emacs.d/site-lisp)
3. Add YASnippet initialization to your .emacs (remember to re-byte-compile it if needed)

```
(require 'yasnippet-bundle)

;;; Bcfg2 snippet

(yas/define-snippets 'sgml-mode
'(
  ("<Bundle" "<Bundle name='${1:bundlename}' version='2.0'>
   $0
</Bundle>" nil)
  ("<Base" "<Base>
   $0
</Base>" nil)
  ("<Group" "<Group name='${1:groupname}>
   $0
</Group>" nil)
  ("<Config" "<ConfigFile name='${1:filename}'/>
$0" nil)
  ("<Service" "<Service name='${1:svcname}'/>
$0" nil)
  ("<Package" "<Package name='${1:packagename}'/>
$0" nil)
  ("<Action" "<Action name='${1:name}'/>
$0" nil)
  ("<Directory" "<Directory name='${1:name}'/>
$0" nil)
  ("<SymLink" "<SymLink name='${1:name}'/>
$0" nil)
  ("<Permissions" "<Permissions name='${1:name}'/>
$0" nil)
)
)
```

4. One quick M-x eval-current-buffer, and this code is enabled

Each of these snippets activates on the opening element, ie <Bundle. After this string is entered, but before entering a space, press <TAB>, and the snippet will be expanded. The template will be inserted into the text

with a set of input prompts, which default to overwrite mode and can be tabbed through.

The code above only works for bundles and base, but will be expanded to support other xml files as well.

8.14 Vim Snippet Support

This page describes using vim with snipMate and a set of snippets that allow quick composition of bundles and base files.

1. Download snipMate from http://www.vim.org/scripts/script.php?script_id=2540
2. Install it using the install instructions (unzip snipMate.zip -d ~/.vim or equivalent, e.g. $HOMEvim-files on Windows)
3. Add the following to `~/.vim/snippets/xml.snippets`

```
# Bundle
snippet <Bundle
        <Bundle name='${1:bundlename}'>
                ${2}
        </Bundle>
# Base
snippet <Base
        <Base>
                ${1}
        </Base>
# Group
snippet <Group
        <Group name='${1:groupname}'>
                ${2}
        </Group>
# ConfigFile
snippet <Config
        <ConfigFile name='${1:filename}'/>
# Service
snippet <Service
        <Service name='${1:svcname}'/>
# Package
snippet <Package
        <Package name='${1:packagename}'/>
# Action
snippet <Action
        <Action name='${1:name}'/>
# Directory
snippet <Directory
        <Directory name='${1:name}'/>
# SymLink
snippet <SymLink
        <SymLink name='${1:name}'/>
# Permissions
snippet <Permissions
        <Permissions name='${1:name}'/>
```

4. Save and start editing away!

Each of these snippets activates on the opening element, ie <Bundle>. After this string is entered, but before entering a space, press <TAB>, and the snippet will be expanded. The template will be inserted into the text with a set of input prompts, which default to overwrite mode and can be tabbed through.

The code above only works for bundles and base, but will be expanded to support other xml files as well.

CHAPTER

NINE

GETTING HELP

Having trouble? We'd like to help!

There are several ways to get in touch with the community around Bcfg2.

- Try the *FAQ* – it's got answers to many common questions.
- Looking for specific information? Try the *genindex*, *modindex*, or the *detailed table of contents*.
- Search for information in the *mailinglist*.
- Ask a question in the *IRC Channel*, or search the IRC logs to see if its been asked before.

Note that the IRC channel tends to be much busier than the mailing list; use whichever seems most appropriate for your query, but don't let the lack of mailing list activity make you think the project isn't active.

9.1 Reporting bugs

Report bugs with Bcfg2 in our tracker.

9.2 Mailing List

To subscribe to the mailing list for Bcfg2 please visit the bcfg-dev mailman page

Searchable archives are available from Gmane. You can also read the mailing list from any NNTP client via Gmane.

9.3 IRC Channel

The Bcfg2 IRC channel is #bcfg2 on Freenode. It is home to both support and development discussions. If you have a question, suggestion, or just want to know about Bcfg2, please drop in and say hi.

Archives are available at: http://colabti.org/irclogger/irclogger_logs/bcfg2

9.4 FAQ

The Frequently Asked Questions (FAQ) answers the most common questions about Bcfg2. At the moment the FAQ is splitted into a general and a client specfic section.

9.4.1 FAQ: General

What does Bcfg2 stand for?

Initially, Bcfg stood for the bundle configuration system. Bcfg2 is the second major revision. At this point, the acronym is meaningless, but the name has stuck. Luckily, Bcfg2 googles better than Bcfg does. No, seriously. Try it. All I know is that I have no interest in a billion cubic feet of gas.

What architectures does Bcfg2 support?

Bcfg2 should run on any POSIX compatible operating system, however direct support for an operating system's package and service formats are limited by the currently available *Available client tools* (although new client tools are pretty easy to add). The following is an incomplete but more exact list of platforms on which Bcfg2 works.

- GNU/Linux deb based distros
- GNU/Linux rpm based distros
- Solaris pkg based
- Gentoo portage based
- OSX (POSIX/launchd support)

What pre-requisites are needed to run Bcfg2?

Please visit the *Prerequisites* section in the manual.

Why won't bcfg2-server start?

If your server doesn't seem to be starting and you see no error messages in your server logs, try running it in the foreground to see why.

Why am I getting a traceback?

If you get a traceback, please let us know. You can file a ticket, send the traceback to the *mailinglist*, or hop on *IRC Channel* and let us know.

What is the most common cause of "The following entries are not handled by any tool"?

Often it corresponds to entries that aren't bound by the server (for which you'll get error messages on the server). You should try inspecting the logs on the server to see what may be the cause.

How can I figure out if error is client or server side?

- Cache a copy of the client using `bcfg2 -c /tmp/config.xml`
- Search for the entry of interest
- If it looks correct, then there is a client issue

This file contains all aspects of client configuration. It is structured as a series of bundles and base entries.

Note: Most often the entry is not correct and the issue lies in the specification.

Where are the server log messages?

The bcfg2-server process logs to syslog facility LOG_DAEMON. The server produces a series of messages upon a variety of events and errors.

Is there a way to check if all repository XML files conform to schemas?

Bcfg2 comes with XML schemas describing all of the XML formats used in the server repository. A validation command `bcfg2-repo-validate` is included with the source distribution and all packages. Run it with the `-v` flag to see each file and the results if its validation.

9.4.2 FAQ: Client

No ca is specified. Cannot authenticate the server with SSL.

This means that the client does not have a copy of the CA for the server, so it can't verify that it is talking to the right server. To fix this issue, copy `/etc/bcfg2.crt` from the server to `/etc/bcfg2.ca` on the client. Then add the following on the client.

```
[communication]
ca = /etc/bcfg2.ca
```

9.5 Error Messages

This page describes error messages produced by Bcfg2 and steps that can be taken to remedy them.

Error	Location	Meaning	Repair
Incomplete information for entry <EntryTag>:<EntryName>; cannot verify	Client	The described entry is not fully specified by the server, so no verification can be performed.	[1]
Incomplete information for entry <EntryTag>:<EntryName>; cannot install	Client	The described entry is not fully specified by the server, so no verification can be performed.	[2]
The following entries are not handled by any tool: <EntryTag>:<EntryName>	Client	The client cannot figure out how to handle this entry.	[3]
no server x509 fingerprint; no server verification performed!	Client	The client is unable to verify the server	[4]
Failed to bind entry: <EntryTag> <EntryName>	Server	The server was unable to find a suitable version of entry for client.	[5]
Failed to bind to socket	Server	The server was unable to bind to the tcp server socket.	[6]
Failed to load ssl key <path>	Server	The server was unable to read and process the ssl key.	[7]
Failed to read file <path>	Server	The server failed to read the specified file	[8]
Failed to parse file <path>	Server	The server failed to parse the specified XML file	[9]
Client metadata resolution error for <IP>	Server	The server cannot resolve the client hostname or the client is associated with a non-profile group.	[10]

[1] This entry is not being bound. Ensure that a version of this entry applies to this client.
[2] This entry is not being bound. Ensure that a version of this entry applies to this client.
[3] Add a type to the generator definition for this entry
[4] Run bcfg2-admin fingerprint on the server and add it to the client bcfg2.conf as mentioned here
[5] This entry is not being bound. Ensure that a version of this entry applies to this client.
[6] Ensure that another instance of the daemon (or any other process) is not listening on the same port.
[7] Ensure that the key is readable by the user running the daemon and that it is well-formed.
[8] Ensure that this file still exists; a frequent cause is the deletion of a temp file.
[9] Ensure that the file is properly formed XML.
[10] Fix hostname resolution for the client or ensure that the profile group is properly setup.

9.6 Manual pages

These are copies of the bcfg2 manual pages created with man2html. The most recent versions of these man pages are always in the *man/* directory in the source.

9.6.1 bcfg2

The man page of bcfg2.

```
man bcfg2
```

9.6.2 bcfg2-admin

The man page of `bcfg2-admin`.

```
man bcfg2-admin
```

9.6.3 bcfg2-build-reports

The man page of `bcfg2-build-reports`.

```
man bcfg2-build-reports
```

9.6.4 bcfg2.conf

The man page of `bcfg2.conf`.

```
man bcfg2.conf
```

9.6.5 bcfg2-info

The man page of `bcfg2-info`.

```
man bcfg2-info
```

9.6.6 bcfg2-repo-validate

The man page of `bcfg2-repo-validate`.

```
man bcfg2-repo-validate
```

9.6.7 bcfg2-server

The man page of `bcfg2-server`.

```
man bcfg2-server
```

9.7 Troubleshooting

From time to time, Bcfg2 produces results that the user finds surprising. This can happen either due to bugs or user error. This page describes several techniques to gain visibility into the bcfg2 client and server and understand what is going on.

9.7.1 Figure out if error is client or server side

- Cache a copy of the client configuration using `bcfg2 -qnc /tmp/config.xml`
- Look in the file and search for the entry of interest
- If it looks correct, then there is a client issue
- If not, it is time to inspect things on the server

This file contains all aspects of client configuration. It is structured as a series of bundles and base entries.

Note: Most often the entry is not correct and the issue lies in the specification.

9.7.2 Review server log messages

The bcfg2-server process logs to syslog facility LOG_DAEMON. The server produces a series of messages upon a variety of events and errors.

9.7.3 Check if all repository XML files conform to schemas

Bcfg2 comes with XML schemas describing all of the XML formats used in the server repository. A validation command `bcfg2-repo-validate` is included with the source distribution and all packages. Run it with the -v flag to see each file and the results if its validation.

9.7.4 If the bcfg2 server is not reflecting recent changes, try restarting the bcfg2-server process

If this fixes the problem, it is either a bug in the underlying file monitoring system (fam or gamin) or a bug in Bcfg2's file monitoring code. In either case, file a ticket in the tracking system. In the ticket, include:

- filesystem monitoring system (fam or gamin)
- kernel version (if on linux)

- if any messages of the form "Handled N events in M seconds" appeared between the modification event and the client configuration generation request appeared in the server log
- which plugin handled the file in the repostiory (Cfg, Rules, Packages, TCheetah, TGenshi, Metadata)
- if a touch of the file after the modification causes the problem to go away

9.7.5 bcfg2-info

Bcfg2 server operations can be simulated using the `bcfg2-info` command. The command is interactive, and has commands to allow several useful operations

- clients - Current client metadata (profile and group) settings
- groups - Current group metadata values
- mappings - Configuration entries provided by plugins
- buildfile <filename> <hostname> - Build a config file for a client
- showentries <client> <type> - Build the abstract configuration (list of entries) for a client
- build <hostname> <output-file> - Build the complete configuration for a client

Type *help* in bcfg2-info for more information.

9.7.6 Error Messages

This page describes error messages produced by Bcfg2 and steps that can be taken to remedy them.

Error	Location	Meaning	Repair
Incomplete information for entry <EntryTag>:<EntryName> cannot verify	Client	The described entry is not fully specified by the server, so no verification can be performed.	[11]
Incomplete information for entry <EntryTag>:<EntryName> cannot install	Client	The described entry is not fully specified by the server, so no verification can be performed.	[1]
The following entries are not handled by any tool: <EntryTag>:<EntryName>	Client	The client cannot figure out how to handle this entry.	[12]
No ca is specified. Cannot authenticate the server with SSL.	Client	The client is unable to verify the server	[13]
Failed to bind entry: <EntryTag> <EntryName>	Server	The server was unable to find a suitable version of entry for client.	[14]
Failed to bind to socket	Server	The server was unable to bind to the tcp server socket.	[15]
Failed to load ssl key <path>	Server	The server was unable to read and process the ssl key.	[16]
Failed to read file <path>	Server	The server failed to read the specified file	[17]
Failed to parse file <path>	Server	The server failed to parse the specified XML file	[18]
Client metadata resolution error for <IP>	Server	The server cannot resolve the client hostname or the client is associated with a non-profile group.	[19]

[11] This entry is not being bound. Ensure that a version of this entry applies to this client.

[12] Add a type to the generator definition for this entry

[13] Copy the Bcfg2 server's CA certificate to the client and specify it using the **ca** option in the [communication] section of `bcfg2.conf`

[14] This entry is not being bound. Ensure that a version of this entry applies to this client.

[15] Ensure that another instance of the daemon (or any other process) is not listening on the same port.

[16] Ensure that the key is readable by the user running the daemon and that it is well-formed.

[17] Ensure that this file still exists; a frequent cause is the deletion of a temp file.

[18] Ensure that the file is properly formed XML.

[19] Fix hostname resolution for the client or ensure that the profile group is properly setup.

9.7.7 FAQs

Why won't bcfg2-server start?

If your server doesn't seem to be starting and you see no error messages in your server logs, try running it in the foreground to see why.

Why am I getting a traceback?

If you get a traceback, please let us know by *reporting it* on Trac, via the mailing list, or on IRC. Your best bet to get a quick response will be to jump on IRC during the daytime (CST).

What is the most common cause of "The following entries are not handled by any tool"?

Often it corresponds to entries that aren't bound by the server (for which you'll get error messages on the server). You should try inspecting the logs on the server to see what may be the cause.

CHAPTER

TEN

GLOSSARY

generator A type of plugin which provides file contents. For example Cfg or TGenshi.

Genshi A Python-based templating engine. Genshi Homepage.

group A "tag" assigned to a client through a probe or other plugin.

irc channel #bcfg2 on freenode

probe A script that executes on a client machine and sets client metadata such as group membership.

profile A special type of group that a client is explicitly assigned to.

repository A collection of folders and files that make up the configurations that Bcfg2 applies to server. The repository is located at `/var/lib/bcfg2` by default. This is not to be confused with a :term:VCS repository, which is an excellent place to pull your Bcfg2 repository from to manage changes. When used alone, repository refers to a Bcfg2 repository.

VCS Stands for Version Control System.

CHAPTER

ELEVEN

APPENDIX

Bcfg2 is based on a client-server architecture. The client is responsible for interpreting (but not processing) the configuration served by the server. This configuration is literal, so no local process is required. After completion of the configuration process, the client uploads a set of statistics to the server. This section will describe the goals and then the architecture motivated by it.

11.1 Example files

In this section are some examples for getting started with a more indeep usage of Bcfg2.

11.1.1 Mysql example

I had some time ago to continue with putting my configuration into Bcfg2 and maybe this helps someone else.

I added a new bundle:

```
<Bundle name="mysql-server" version="3.0">
   <ConfigFile name="/root/bcfg2-install/mysql/users.sh"/>
   <ConfigFile name="/root/bcfg2-install/mysql/users.sql"/>
   <PostInstall name="/root/bcfg2-install/mysql/users.sh"/>
   <Package name="mysql-server-4.1"/>
   <Service name="mysql"/>
</Bundle>
```

The `users.sh` script looks like this:

```
#!/bin/sh

mysql --defaults-extra-file=/etc/mysql/debian.cnf mysql \
    < /root/bcfg2-install/mysql/users.sql
```

On debian there is a user account in `/etc/mysql/debian.cnf` automatically created, but you could also (manually) create a user in the database that has enough permissions and add the login information in a file yourself. This file looks like this:

```
[client]
host     = localhost
user     = debian-sys-maint
password = XXXXXXXXXX
```

The `users.sql` looks like this:

```
DELETE FROM db;
INSERT INTO db VALUES ('localhost', 'phpmyadmin', 'pma', 'Y', 'Y',
'Y', 'Y', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N');

DELETE FROM user WHERE User <> 'debian-sys-maint';
INSERT INTO user VALUES ('localhost', 'root', 'XXXXXXXXXXX', 'Y', 'Y',
'Y', 'Y', 'Y', 'Y', 'Y', 'Y', 'Y', 'Y', 'Y', 'Y', 'Y', 'Y', 'Y', 'Y',
'Y', 'Y', 'Y', 'Y', 'Y', '', '', '', '', 0, 0, 0);
INSERT INTO user VALUES ('localhost', 'pma', '', 'N', 'N', 'N', 'N',
'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N',
'N', 'N', 'N', '', '', '', '', 0, 0, 0);

FLUSH PRIVILEGES;
```

11.1.2 ntp example

Here is a series of example configurations for Bcfg2, each introducing another layer of functionality.

- After each change, run `bcfg-repo-validate -v`
- Run the server with `bcfg2-server -v`
- Update the client with `bcfg2 -v -d -n` (will not actually make client changes)

Package only

Our example starts with the bare minimum configuration setup. We have a client, a profile group, a list of packages, and a base configuration.

```
# cat Metadata/clients.xml
<Clients version='3.0'>
<Client profile='fedora' pingable='N' pingtime='0' name='foo.bar.com'/>
</Clients>

# cat Metadata/groups.xml
<Groups version='3.0'>
<Group profile='true' name='fedora' toolset='rh'/>
</Groups>

# cat Base/base.xml
<Base>
<Group name='fedora'>
<Package name='ntp'/>
</Group>
</Base>
```

```
# cat Pkgmgr/packages.xml
<PackageList type='rpm' priority='0'>
<Package name='ntp' version='4.2.0.a.20050816-11.FC5'/>
</PackageList>
```

Add service

Configure the service, and add it to the base.

```
# cat Svcmgr/services.xml
<Services priority='0'>
<Service name='ntpd' status='on'/>
</Services>

# cat Base/base.xml
<Base>
<Group name='fedora'>
<Package name='ntp'/>
<Service name='ntpd'/>
</Group>
</Base>
```

Add config file

Setup an `etc/` directory structure, and add it to the base.

```
# cat Cfg/etc/ntp.conf/ntp.conf
server ntp1.utexas.edu

 # cat Base/base.xml
 <Base>
 <Group name='fedora'>
 <Package name='ntp'/>
 <Service name='ntpd'/>
 <ConfigFile name='/etc/ntp.conf'/>
 </Group>
</Base>
```

Create a bundle

The above configuration layout works fine for a single service, but that method of organization would quickly become a nightmare as you approach the number of packages, services, and config files required to represent a fully configured host. Bundles allow the grouping of related configuration entries that are used to provide a single service. This is done for several reasons:

- Grouping related things in one place makes it easier to add those entries for a multiple groups of clients

- Grouping entries into bundles makes their validation occur collectively. This means that config files can override the contents of packages. Also, config files are rechecked after packages are upgraded, so that they can be repaired if the package install clobbered them.
- Services associated with a bundle get restarted whenever any entity in that bundle is modified. This ensures that new configuration files and software are used after installation.

The config file, package, and service are really all related components describing the idea of an ntp client, so they should be logically grouped together. We use a bundle to accomplish this.

```
# cat Bundler/ntp.xml
<Bundle name='ntp' version='2.0'>
<Package name='ntp'/>
<Service name='ntpd'/>
<ConfigFile name='/etc/ntp.conf'/>
</Bundle>
```

After this bundle is created, it must be associated with a group (or groups). Add a bundle child element to the group(s) which should install this bundle.

```
 # cat Metadata/groups.xml
 <Groups>
...
 <Group name='fedora'>
   <Bundle name='ntp'/>
 </Group>
...
 </Groups>
```

Once this bundle is created, a client reconfigure will install these entries. If any are modified, then the `ntpd` service will be restarted. If you only want ntp configurations to be updated (and nothing else), the bcfg2 client can be run with a `-b <bundle name>` option that will only update entries in the specified bundle.

11.2 Example configuration

This section contains useful configuration of additional tools.

11.2.1 mrepo

This section describes how to setup an mrepo mirror.

mrepo builds a local APT/Yum RPM repository from local ISO files, downloaded updates, and extra packages from 3rd party repositories. It takes care of setting up the ISO files, downloading the RPMs, configuring HTTP access and providing PXE/TFTP resources for remote network installations.

Sample mrepo configuration

```
### Configuration file for mrepo

### The [main] section allows to override mrepo's default settings
### The mrepo-example.conf gives an overview of all the possible settings
[main]
srcdir = /var/mrepo/src
wwwdir = /var/www/mrepo
confdir = /etc/mrepo.conf.d
arch = x86_64

mailto = <youremail>
smtp-server = localhost

hardlink = yes
shareiso = yes

rsync-timeout = 3600

[centos5]
name = CentOS Server $release ($arch)
release = 5
arch = x86_64
metadata = yum repomd

# ISO images
iso = centos-$release-server-$arch-DVD.iso

#addons = rsync://mirrors.kernel.org/centos/$release/addons/$arch/RPMS
centosplus = rsync://mirrors.kernel.org/centos/$release/centosplus/$arch/RPMS
extras = rsync://mirrors.kernel.org/centos/$release/extras/$arch/RPMS
#fasttrack = rsync://mirrors.kernel.org/centos/$release/fasttrack/$arch/RPMS
os = rsync://mirrors.kernel.org/centos/$release/os/$arch/CentOS
updates = rsync://mirrors.kernel.org/centos/$release/updates/$arch/RPMS
dag = http://apt.sw.be/redhat/el$release/en/$arch/RPMS.dag
dries = http://apt.sw.be/redhat/el$release/en/$arch/RPMS.dries
rpmforge = http://apt.sw.be/redhat/el$release/en/$arch/RPMS.rpmforge

### Any other section is considered a definition for a distribution
### You can put distribution sections in /etc/mrepo.conf.d/
### Examples can be found in the documentation at:
###     /usr/share/doc/mrepo-0.8.6/dists/.
```

Update the repositories

To update your local repository, just lauch the following command

```
mrepo -ug
```

11.3 Contributors

In alphabetical order of the given name:

- Brian Pellin and Andrew Lusk did substantial work on Bcfg1, some of which was used in the bcfg2 client.
- Chris Vuletich <vuletich@mcs.anl.gov> wrote some SSL code and the verification debugging code
- Cory Lueninghoener <cory@mcs.anl.gov> wrote the showentries function in `bcfg2-info`
- Daniel Clark <dclark@pobox.com> created encap packages for bcfg2 and deps, wrote fossil-scm dvcs support, and helps with Debian packaging
- Danny Clark enabled the Encap packaging
- David Dahl worked on Hostbase
- David Strauss worked on CentOS, RHEL, Yum, and Bazaar VCS support
- Ed Smith <esmith4@inf.ed.ac.uk> has done substantial hardening of the bcfg client and server and implemented a common logging infrastructure.
- Fabian Affolter <fabian@bernewireless.net> made some patches and worked on the manual
- Jason Pepas <cell@ices.utexas.edu> has written a RPM package list creator has contributed patches to the Red Hat toolset
- Joey Hagedorn <hagedorn@mcs.anl.gov> has written the reporting subsystem, including StatReports, GenerateHostinfo, and the xslt, css and javascript associated with it.
- Jos Catnook fixed bugs
- Ken Raffenetti <raffenet@mcs.anl.gov> and Rick Bradshaw have written the Hostbase plugin
- Michael Jinks <mjinks@uchicago.edu> wrote the Gentoo tool drivers
- Narayan Desai <desai@mcs.anl.gov> has written most of bcfg2, including all parts not explicitly mentioned in this file
- Patrick Ruckstuhl fixed bugs in the templating
- Pedro Flores made the Reporting system design help
- Rick Bradshaw <bradshaw@mcs.anl.gov> has written several of the tools included in the `tools/` subdirectory
- Sami Haahtinen <ressu@ressukka.net> has written Debian packaging logic.
- Scott Behrens <behrens@mcs.anl.gov> and Rick Bradshaw have written the VHost plugin
- Scott Matott
- Sol Jerome <solj@ices.utexas.edu> squashes bugs, helps manage the project roadmap, and implements various interesting features.

- Ti Leggett worked on ebuild packaging and bugfixes, RPM packaging
- Zach Lowry Solaris support and general hardening

The entire MCS systems team has provided invaluable help in the design process and refinement of the user interface. In particular, Gene Rackow and Sandra Bittner have provided great assistance throughout this project. Philip Steinbachs provided detailed feedback as an early external user.

The most updated listing is available in the AUTHORS file in the SVN *repository* of Bcfg2.

11.4 Books

- Configuration Management with Bcfg2
- Narayan Desai and Cory Lueninghoener

11.5 Papers

- Configuration Life-Cycle Management on the TeraGrid.
- Ti Leggett, Cory Lueninghoener, and Narayan Desai
- In Proceedings of TeraGrid '07 Conference, June 2007
- A Scalable Approach To Deploying And Managing Appliances
- Rick Bradshaw, Narayan Desai, Tim Freeman, and Kate Keahey
- In Proceedings of the TeraGrid '07 Conference, June 2007
- Bcfg2 - Konfigurationsmanagement Für Heterogene Umgebungen
- Marko Jung, Robert Gogolok
- In Proceedings of German Unix User Group's Frühjahrsfachgespräch 2007, March 2007.
- Directing Change Using Bcfg2
- Narayan Desai, Rick Bradshaw, Joey Hagedorn, and Cory Lueninghoener
- In Proceedings of the Twentieth Large Install System Administration Conference (LISA XX), December 2-9, 2006, Washington D.C., USA, 2006.
- A Case Study in Configuration Management Tool Deployment
- Narayan Desai, Rick Bradshaw, Scott Matott, Sandra Bittner, Susan Coghlan, Remy Evard, Cory Leunighhoener, Ti Leggett, J.P. Navarro, Gene Rackow, Craig Stacey, and Tisha Stacey
- In Proceedings of the Nineteenth Large Install System Administration Conference (LISA XIX), December 4-9, 2005, San Diego, CA, USA, 2005.
- Bcfg2: A Pay As You Go Approach to Configuration Complexity
- Narayan Desai

- In Proceedings of the 2005 Australian Unix Users Group (AUUG2005), October 16-21, 2005, Sydney, Australia, 2005.
- Bcfg: A Configuration Management Tool for Heterogenous Environments
- Narayan Desai, Andrew Lusk, Rick Bradshaw, and Remy Evard
- In Proceedings of the 5th IEEE International Conference on Cluster Computing (CLUSTER03), pages 500-503. IEEE Computer Society, 2003.

11.6 Articles

- Configuration and change management with Bcfg2: "The Dean" - The powerful Bcfg2 provides a sophisticated environment for centralized configuration management.
- Marko Jung, Nils Magnus
- In the english 'Linux Magazine', 04/09, pages 30-35, April 2009
- The Bcfg2 code listings for the article are public.
- Konfigurations- und Change-Management in Bcfg2
- Marko Jung, Nils Magnus
- In the german 'Linux Magazin', 10/08, pages 76-80, September 2008
- The code listings for the article are public.
- System Management Methodologies with Bcfg2
- Narayan Desai, Rick Bradshaw and Joey Hagedorn
- In ;login: Magazine, Volume 31, #1, pages 11-18, February 2006

11.7 Guides

This section contains platform-specific quickstart guides and howtos around Bcfg2.

11.7.1 Authentication

Scenarios

1. Cluster nodes that are frequently rebuilt

   Default settings work well; machines do not float, and a per-client password is not required.

2. *NAT Howto nat_howto*

- Build client records in advance with `bcfg2-admin`, setting a uuid for each new client.
- Set the address attribute for each to the address of the NAT.

- Optionally, set a per-client password for each, and set into secure mode.

  Note: This will require the use of the uuid and password from each client, and will require that they come through the NAT address.

Building bcfg2.conf automatically

This is a TCheetah template that automatically constructs per-client *bcfg2.conf* from the per-client metadata:

```
[communication]
protocol = xmlrpc/ssl
#if $self.metadata.uuid != None
user = $self.metadata.uuid
#end if
#if $self.metadata.password != None
password = $self.metadata.password
#else
password = my-password-foobar
#end if

[components]
bcfg2 = https://localhost:6789
```

In this setup, this will cause any clients that have uuids established to be set to use them in *bcfg2.conf*. It will also cause any clients with passwords set to use them instead of the global password.

How Authentication Works

1. First, the client is associated with a client record. If the client specifies a uuid, it uses this instead of the results of a dns or address lookup.
2. Next, the ip address is verified against the client record. If the address doesn't match, then the client must be set to location=floating
3. Finally, the password is verified. If the client is set to secure mode, the only its per-client password is accepted. If it is not set to secure mode, then either the global password or per-client password will be accepted

Failure during any of these stages results in authentication failure. Note that clients set into secure mode that do not have per-client passwords set will not be able to connect.

SSL Cert-based client authentication

SSL-based client authentication is supported. This requires several things:

1. Certificate Authority (to sign all keys)
2. Server key and cert signed by the CA
3. Client key and cert signed by the CA

A variety of CAs can be used, but these keys can be simply generated using the following set of steps:

1. Setup a CA

   http://www.flatmtn.com/article/setting-openssl-create-certificates

2. Create keys for each client and server, signing them with the CA signing cert

   http://www.flatmtn.com/article/setting-ssl-certificates-apache

   Note: The client CN must be the FQDN of the client (as returned by a reverse DNS lookup of the ip address. Otherwise, you will end up with an error message on the client that looks like:

   ```
   Server failure: Protocol Error: 401 Unauthorized
   Failed to download probes from bcfg2
   Server Failure
   ```

   You will also see an error message on the server that looks something like:

   ```
   cmssrv01 bcfg2-server[9785]: Got request for cmssrv115 from incorrect address 131.22
   cmssrv01 bcfg2-server[9785]: Resolved to cmssrv115.fnal.gov
   ```

3. Distribute the keys and certs to the appropriate locations

4. Copy the ca cert to clients, so that the server can be authenticated

Clients authenticating themselves with a certificate will be authenticated that way first; clients can be setup to either authenticate solely with certs, use certs with a fallback to password, or password only. Also a bootstrap mode will be added shortly; this will allow a client to authenticate with a password its first time, requiring a certificate all subsequent times. This behavior can be controlled through the use of the auth attribute in *Metadata/clients.xml*:

```
<Clients>
  <Client name='testclient' auth='cert'/>
</Clients>
```

Allowed values are:

Auth Type	Meaning
cert	Certificates must be used
cert+password	Certificate or password may be used
bootstrap	Password can be used for one client run, after that certificate is required

11.7.2 Quickstart for CentOS

This is a complete getting started guide for CentOS. With this document you should be able to install a Bcfg2 server and a Bcfg2 client.

Install Bcfg2

The fastest way to get Bcfg2 onto your system is to use Yum or your preferred package management tool. We'll be using the ones that are distributed through EPEL, but depending on your aversion to risk you could download an RPM from other places as well. See *Using Bcfg2 With CentOS* for information about building Bcfg2 from source and making your own packages.

Using EPEL

Make sure EPEL is a valid repository on your server. The instructions on how to do this basically say:

```
[root@centos ~]# rpm -Uvh http://download.fedora.redhat.com/pub/epel/5/x86_64/epel-release-
```

Note: You will have to adjust this command to match your architecture and the current EPEL release.

Install the bcfg2-server and bcfg2 RPMs:

```
[root@centos ~]# yum install bcfg2-server bcfg2
```

Your system should now have the necessary software to use Bcfg2. The next step is to set up your Bcfg2 *repository*.

Initialize your repository

Now that you're done with the install, you need to initialize your repository and setup your `/etc/bcfg2.conf`. `bcfg2-admin init` is a tool which allows you to automate this:

```
[root@centos ~]# bcfg2-admin init
Store bcfg2 configuration in [/etc/bcfg2.conf]:
Location of bcfg2 repository [/var/lib/bcfg2]:
Input password used for communication verification (without echoing; leave blank for a ran
What is the server's hostname: [centos]
Input the server location [https://centos:6789]:
Input base Operating System for clients:
1: Redhat/Fedora/RHEL/RHAS/Centos
2: SUSE/SLES
3: Mandrake
4: Debian
5: Ubuntu
6: Gentoo
7: FreeBSD
: 1
Generating a 2048 bit RSA private key
.........................+++
...................+++
writing new private key to '/etc/bcfg2.key'
-----
Signature ok
subject=/C=US=ST=Illinois/L=Argonne/CN=centos
Getting Private key
Repository created successfuly in /var/lib/bcfg2
```

Change responses as necessary.

Start the server

You are now ready to start your bcfg2 server for the first time:

```
[root@centos ~]# /sbin/service bcfg2-server start
```

To verify that everything started ok, look for the running daemon and check the logs:

```
[root@centos ~]# /etc/init.d/service bcfg2-server status
[root@centos ~]# tail /var/log/messages
Mar 29 12:42:26 centos bcfg2-server[5093]: service available at https://centos:6789
Mar 29 12:42:26 centos bcfg2-server[5093]: serving bcfg2-server at https://centos:6789
Mar 29 12:42:26 centos bcfg2-server[5093]: serve_forever() [start]
Mar 29 12:42:41 centos bcfg2-server[5093]: Handled 16 events in 0.007s
```

Run bcfg2 to be sure you are able to communicate with the server:

```
[root@centos ~]# bcfg2 -vqn
No ca is specified. Cannot authenticate the server with SSL.
No ca is specified. Cannot authenticate the server with SSL.
Loaded plugins: fastestmirror
Loading mirror speeds from cached hostfile
Excluding Packages in global exclude list
Finished
Loaded tool drivers:
 Action       Chkconfig  POSIX        YUMng

Phase: initial
Correct entries:        0
Incorrect entries:      0
Total managed entries:  0
Unmanaged entries:      208


Phase: final
Correct entries:        0
Incorrect entries:      0
Total managed entries:  0
Unmanaged entries:      208

No ca is specified. Cannot authenticate the server with SSL.
```

The ca message is just a warning, meaning that the client does not have sufficient information to verify that it is talking to the correct server. This can be fixed by distributing the ca certificate from the server to all clients. By default, this file is available in `/etc/bcfg2.crt` on the server. Copy this file to the client (with a bundle) and add the ca option to `bcfg2.conf` pointing at the file, and the client will be able to verify it is talking to the correct server upon connection:

```
[root@centos ~]# cat /etc/bcfg2.conf


[communication]
protocol = xmlrpc/ssl
password = N41lMNeW
ca = /etc/bcfg2.crt

[components]
```

```
bcfg2 = https://centos:6789
```

Now if you run the client, no more warning:

```
[root@centos ~]# bcfg2 -vqn
Loaded plugins: fastestmirror
Loading mirror speeds from cached hostfile
Excluding Packages in global exclude list
Finished
Loaded tool drivers:
 Action       Chkconfig  POSIX        YUMng

Phase: initial
Correct entries:        0
Incorrect entries:      0
Total managed entries:  0
Unmanaged entries:      208


Phase: final
Correct entries:        0
Incorrect entries:      0
Total managed entries:  0
Unmanaged entries:      208
```

Bring your first machine under Bcfg2 control

Now it is time to get your first machine's configuration into your Bcfg2 *repository*. Let's start with the server itself.

Setup the Packages plugin

First, replace **Pkgmgr** with **Packages** in the plugins line of `bcfg2.conf`. Then create Packages layout (as per *Example usage*) in `/var/lib/bcfg2`

Note: I am using the RawURL syntax here since we are using mrepo to manage our yum mirrors.

```
<Sources>
        <!-- CentOS (5.4) sources -->
        <YUMSource>
                <Group>centos5.4</Group>
                <RawURL>http://mrepo/centos5-x86_64/RPMS.os</RawURL>
                <Arch>x86_64</Arch>
        </YUMSource>
        <YUMSource>
                <Group>centos5.4</Group>
                <RawURL>http://mrepo/centos5-x86_64/RPMS.updates</RawURL>
                <Arch>x86_64</Arch>
        </YUMSource>
        <YUMSource>
                <Group>centos5.4</Group>
```

```
                <RawURL>http://mrepo/centos5-x86_64/RPMS.extras</RawURL>
                <Arch>x86_64</Arch>
        </YUMSource>
</Sources>
```

Due to the Magic Groups, we need to modify our Metadata. Let's add a **centos5.4** group which inherits a **centos** group (this should replace the existing **redhat** group) present in `/var/lib/bcfg2/Metadata/groups.xml`. The resulting file should look something like this

```
<Groups version='3.0'>
   <Group profile='true' public='true' default='true' name='basic'>
      <Group name='centos5.4'/>
   </Group>
   <Group name='centos5.4'>
      <Group name='centos'/>
   </Group>
   <Group name='ubuntu'/>
   <Group name='debian'/>
   <Group name='freebsd'/>
   <Group name='gentoo'/>
   <Group name='centos'/>
   <Group name='suse'/>
   <Group name='mandrake'/>
   <Group name='solaris'/>
</Groups>
```

Note: When editing your xml files by hand, it is useful to occasionally run *bcfg2-repo-validate* to ensure that your xml validates properly.

The final thing we need is for the client to have the proper arch group membership. For this, we will make use of the *unsorted-dynamic_groups* capabilities of the Probes plugin. Add Probes to your plugins line in `bcfg2.conf` and create the Probe.:

```
[root@centos ~]# grep plugins /etc/bcfg2.conf
plugins = Base,Bundler,Cfg,Metadata,Packages,Probes,Rules,SSHbase
[root@centos ~]# mkdir /var/lib/bcfg2/Probes
[root@centos ~]# cat /var/lib/bcfg2/Probes/groups
#!/bin/sh

echo "group:`uname -m`"
```

Now we restart the bcfg2-server:

```
[root@centos ~]# /etc/init.d/bcfg2-server restart
```

If you tail `/var/log/syslog` now, you will see the Packages plugin in action, updating the cache.

Start managing packages

Add a base-packages bundle. Let's see what happens when we just populate it with the *yum* package.

```
[root@centos ~]# cat /var/lib/bcfg2/Bundler/base-packages.xml
<Bundle name='base-packages'>
        <Package name='yum'/>
</Bundle>
```

You need to reference the bundle from your Metadata. The resulting profile group might look something like this

```
<Group profile='true' public='true' default='true' name='basic'>
   <Bundle name='base-packages'/>
   <Group name='centos5.4'/>
</Group>
```

Now if we run the client, we can see what this has done for us.:

```
[root@centos ~]# bcfg2 -vqn
Running probe groups
Probe groups has result:
x86_64
Loaded plugins: fastestmirror
Loading mirror speeds from cached hostfile
Excluding Packages in global exclude list
Finished
Loaded tool drivers:
 Action       Chkconfig  POSIX        YUMng
        Package pam failed verification.

Phase: initial
Correct entries:        94
Incorrect entries:      1
Total managed entries:  95
Unmanaged entries:      113

In dryrun mode: suppressing entry installation for:
  Package:pam

Phase: final
Correct entries:        94
Incorrect entries:      1
 Package:pam
Total managed entries:  95
Unmanaged entries:      113
```

Interesting, our **pam** package failed verification. What does this mean? Let's have a look:

```
[root@centos ~]# rpm --verify pam
....L...  c /etc/pam.d/system-auth
```

Sigh, it looks like the default RPM install for pam fails to verify using its own verification process (trust me, it's not the only one). At any rate, I was able to get rid of this particular issue by removing the symlink and running `yum reinstall pam`.

As you can see, the Packages plugin has generated the dependencies required for the yum package automatically. The ultimate goal should be to move all the packages from the **Unmanaged** entries section to the

Managed entries section. So, what exactly *are* those Unmanaged entries?:

```
[root@centos ~]# bcfg2 -veqn
Running probe groups
Probe groups has result:
x86_64
Loaded plugins: fastestmirror
Loading mirror speeds from cached hostfile
Excluding Packages in global exclude list
Finished
Loaded tool drivers:
 Action       Chkconfig  POSIX        YUMng
Extra Package openssh-clients 4.3p2-36.el5_4.4.x86_64.
Extra Package libuser 0.54.7-2.1el5_4.1.x86_64.
...

Phase: initial
Correct entries:        95
Incorrect entries:      0
Total managed entries:  95
Unmanaged entries:      113


Phase: final
Correct entries:        95
Incorrect entries:      0
Total managed entries:  95
Unmanaged entries:      113
 Package:at
 Package:avahi
 Package:avahi-compat-libdns_sd
 ...
```

Now you can go through these and continue adding the packages you want to your Bundle. After a while, I ended up with a minimal bundle that looks like this

```
<Bundle name='base-packages'>
        <Package name='bcfg2-server'/>
        <Package name='exim'/>
        <Package name='grub'/>
        <Package name='kernel'/>
        <Package name='krb5-workstation'/>
        <Package name='m2crypto'/>
        <Package name='openssh-clients'/>
        <Package name='openssh-server'/>
        <Package name='prelink'/>
        <Package name='redhat-lsb'/>
        <Package name='rpm-build'/>
        <Package name='rsync'/>
        <Package name='sysklogd'/>
        <Package name='vim-enhanced'/>
        <Package name='yum'/>
</Bundle>
```

Now when I run the client, you can see I have only one unmanaged package:

```
[root@centos ~]# bcfg2 -veqn
Running probe groups
Probe groups has result:
x86_64
Loaded plugins: fastestmirror
Loading mirror speeds from cached hostfile
Excluding Packages in global exclude list
Finished
Loaded tool drivers:
 Action       Chkconfig  POSIX        YUMng
Extra Package gpg-pubkey e8562897-459f07a4.None.
Extra Package gpg-pubkey 217521f6-45e8a532.None.

Phase: initial
Correct entries:        187
Incorrect entries:      0
Total managed entries:  187
Unmanaged entries:      16


Phase: final
Correct entries:        187
Incorrect entries:      0
Total managed entries:  187
Unmanaged entries:      16
 Package:gpg-pubkey
 Service:atd
 Service:avahi-daemon
 Service:bcfg2-server
 ...
```

The gpg-pubkey packages are special in that they are not really packages. Currently, the way to manage them is using *BoundEntries*. So, after adding them, our Bundle now looks like this

Note: This does not actually control the contents of the files, you will need to do this part separately (see below).

```
<Bundle name='base-packages'>
        <BoundPackage name="gpg-pubkey" type="rpm">
                        <Instance simplefile="/etc/pki/rpm-gpg/RPM-GPG-KEY-CentOS-5" versi
                        <Instance simplefile="/etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL" version="
        </BoundPackage>
        <Package name='bcfg2-server'/>
        <Package name='exim'/>
        <Package name='grub'/>
        <Package name='kernel'/>
        <Package name='krb5-workstation'/>
        <Package name='m2crypto'/>
        <Package name='openssh-clients'/>
        <Package name='openssh-server'/>
        <Package name='prelink'/>
        <Package name='redhat-lsb'/>
```

```
        <Package name='rpm-build'/>
        <Package name='rsync'/>
        <Package name='sysklogd'/>
        <Package name='vim-enhanced'/>
        <Package name='yum'/>
</Bundle>
```

To actually push the gpg keys out via Bcfg2, you will need to manage the files as well. This can be done by adding Path entries for each of the gpg keys you want to manage

```
<Bundle name='base-packages'>
        <Path name='/etc/pki/rpm-gpg/RPM-GPG-KEY-CentOS-5'/>
        <Path name='/etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL'/>
        <BoundPackage name="gpg-pubkey" type="rpm">
                        <Instance simplefile="/etc/pki/rpm-gpg/RPM-GPG-KEY-CentOS-5" versi
                        <Instance simplefile="/etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL" version="
        </BoundPackage>
        <Package name='bcfg2-server'/>
        <Package name='exim'/>
        <Package name='grub'/>
        <Package name='kernel'/>
        <Package name='krb5-workstation'/>
        <Package name='m2crypto'/>
        <Package name='openssh-clients'/>
        <Package name='openssh-server'/>
        <Package name='prelink'/>
        <Package name='redhat-lsb'/>
        <Package name='rpm-build'/>
        <Package name='rsync'/>
        <Package name='sysklogd'/>
        <Package name='vim-enhanced'/>
        <Package name='yum'/>
</Bundle>
```

Then add the files to Cfg:

```
mkdir -p Cfg/etc/pki/rpm-gpg/RPM-GPG-KEY-CentOS-5
cp /etc/pki/rpm-gpg/RPM-GPG-KEY-CentOS-5 !$/RPM-GPG-KEY-CentOS-5
mkdir -p Cfg/etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL
cp /etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL !$/RPM-GPG-KEY-EPEL
```

Now, running the client shows only unmanaged Service entries. Woohoo!

Manage services

Now let's clear up the unmanaged service entries by adding the following entries to our bundle.

```
<!-- basic services -->
<Service name='atd'/>
<Service name='avahi-daemon'/>
<Service name='bcfg2-server'/>
<Service name='crond'/>
```

```
<Service name='cups'/>
<Service name='gpm'/>
<Service name='lvm2-monitor'/>
<Service name='mcstrans'/>
<Service name='messagebus'/>
<Service name='netfs'/>
<Service name='network'/>
<Service name='postfix'/>
<Service name='rawdevices'/>
<Service name='sshd'/>
<Service name='syslog'/>
```

...and bind them in Rules

```
[root@centos ~]# cat /var/lib/bcfg2/Rules/services.xml
<Rules priority='1'>
        <!-- basic services -->
        <Service type='chkconfig' status='on' name='atd'/>
        <Service type='chkconfig' status='on' name='avahi-daemon'/>
        <Service type='chkconfig' status='on' name='bcfg2-server'/>
        <Service type='chkconfig' status='on' name='crond'/>
        <Service type='chkconfig' status='on' name='cups'/>
        <Service type='chkconfig' status='on' name='gpm'/>
        <Service type='chkconfig' status='on' name='lvm2-monitor'/>
        <Service type='chkconfig' status='on' name='mcstrans'/>
        <Service type='chkconfig' status='on' name='messagebus'/>
        <Service type='chkconfig' status='on' name='netfs'/>
        <Service type='chkconfig' status='on' name='network'/>
        <Service type='chkconfig' status='on' name='postfix'/>
        <Service type='chkconfig' status='on' name='rawdevices'/>
        <Service type='chkconfig' status='on' name='sshd'/>
        <Service type='chkconfig' status='on' name='syslog'/>
</Rules>
```

Now we run the client and see there are no more unmanaged entries!

```
[root@centos ~]# bcfg2 -veqn
Running probe groups
Probe groups has result:
x86_64
Loaded plugins: fastestmirror
Loading mirror speeds from cached hostfile
Excluding Packages in global exclude list
Finished
Loaded tool drivers:
 Action       Chkconfig  POSIX        YUMng

Phase: initial
Correct entries:        205
Incorrect entries:      0
Total managed entries:  205
Unmanaged entries:      0
```

```
Phase: final
Correct entries:        205
Incorrect entries:      0
Total managed entries:  205
Unmanaged entries:      0
```

Dynamic (web) reports

See installation instructions at *server-reports-install*

11.7.3 Converging on Verification with RHEL 5

Running verification

To get complete verification status, run:

```
bcfg2 -vqned
```

Unmanaged entries

- Package (top-level)

1. Enable the "Packages" plugin in {{{/etc/bcfg2.conf}}}, and configure the Yum repositories in {{{/var/lib/bcfg2/Packages/config.xml}}}.
2. If a package is unwanted, remove it:

   ```
   sudo yum remove PACKAGE
   ```

3. Otherwise, add {{{<Package name="PACKAGE" />}}} to the Base or Bundler configuration.

- Package (dependency)

1. Ensure the Yum repository sources configured in {{{/var/lib/bcfg2/Packages/config.xml}}} are correct.
2. Ensure the Yum repositories themselves are up-to-date with the main package and dependencies.
3. Rebuild the Packages plugin cache:

   ```
   bcfg2-admin xcmd Packages.Refresh
   ```

- Service

1. Add {{{<Service name="SERVICE" />}}} to the Base or Bundler configuration.
2. Add {{{<Service name="SERVICE" status="on" type="chkconfig" />}}} to {{{/var/lib/bcfg2/Rules/services.xml}}}.

Incorrect entries

For a "Package"

- Failed RPM verification

1. Run {{{rpm -V PACKAGE}}}
2. Add configuration files (the ones with "c" next to them in the verification output) to {{{/var/lib/bcfg2/Cfg/}}}.

- For example, {{{/etc/motd}}} to {{{/var/lib/bcfg2/Cfg/etc/motd/motd}}}. Yes, there is an extra directory level named after the file.

1. Specify configuration files as {{{<Path name='PATH' />}}} in the Base or Bundler configuration.
2. Add directories to {{{/var/lib/bcfg2/Rules/directories.xml}}}. For example:

```
<Rules priority="0">
  <Directory name="/etc/cron.hourly" group="root" owner="root" perms="0700"
  <Directory name="/etc/cron.daily" group="root" owner="root" perms="0700" /
</Rules>
```

- Multiple instances

- Option A: Explicitly list the instances

1. Drop the {{{<Package />}}} from the Base or Bundler configuration.
2. Add an explicit {{{<BoundPackage>}}} and {{{<Instance />}}} configuration to a new Bundle, like the following:

```
<Bundle name='keys'>
  <!-- GPG keys -->
  <BoundPackage name="gpg-pubkey" type="yum">
    <Instance simplefile="/etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL" version="21752
    <Instance simplefile="/etc/pki/rpm-gpg/RPM-GPG-KEY-redhat-release" vers
  </BoundPackage>
</Bundle>
```

3. Add the bundle to the applicable groups in {{{/var/lib/bcfg2/Metadata/groups.xml}}}.

- Option B: Disable verification of the package

1. Add {{{pkg_checks="false"}}} to the {{{<Package />}}} tag.

For a "Path"

- Unclear verification problem (no details from BCFG2)

1. Run {{{bcfg2 -vqI}}} to see detailed verification issues (but deny any suggested actions).

- Permissions mismatch

1. Create an {{{info.xml}}} file in the same directory as the configuration file. Example:

```
<FileInfo>
  <Group name='webserver'>
    <Info owner='root' group='root' perms='0652'/>
  </Group>
  <Info owner='root' group='sys' perms='0651'/>
</FileInfo>
```

Other troubleshooting tools

- Generate the physical configuration from the server side:

  ```
  bcfg2-info buildfile /test test.example.com
  ```

- Generate the physical configuration from the client side:

  ```
  bcfg2 -vqn -c/root/bcfg2-physical.xml
  ```

11.7.4 Fedora

This guide is work in progess.

This is a complete getting started guide for Fedora. With this document you should be able to install a Bcfg2 server, a Bcfg2 client, and change the `/etc/motd` file on the client.

Install Bcfg2 From RPM

The fastest way to get Bcfg2 onto your system is to use `yum` or PackageKit. ‘‘um‘‘ will pull all dependencies of Bcfg2 automatically in.

```
$ su -c 'yum install bcfg2-server bcfg2'
```

Your system should now have the necessary software to use Bcfg2. The next step is to set up your Bcfg2 *repository*.

Initialize your repository

Now that you're done with the install, you need to initialize your repository and setup your `/etc/bcfg2.conf`. `bcfg2-admin init` is a tool which allows you to automate this:

```
# bcfg2-admin init
Store bcfg2 configuration in [/etc/bcfg2.conf]:
Location of bcfg2 repository [/var/lib/bcfg2]:
Directory /var/lib/bcfg2 exists. Overwrite? [y/N]:y
Input password used for communication verification (without echoing; leave blank for a ran
What is the server's hostname: [config01.local.net]
Input the server location [https://config01.local.net:6789]:
Input base Operating System for clients:
```

```
1: Redhat/Fedora/RHEL/RHAS/Centos
2: SUSE/SLES
3: Mandrake
4: Debian
5: Ubuntu
6: Gentoo
7: FreeBSD
: 1
Generating a 1024 bit RSA private key
.....................................................++++++
.....++++++
writing new private key to '/etc/bcfg2.key'
-----
Signature ok
subject=/C=US/ST=Illinois/L=Argonne/CN=config01.local.net
Getting Private key
Repository created successfuly in /var/lib/bcfg2
```

Change responses as necessary.

Start the server

You are now ready to start your bcfg2 server for the first time:

```
$ su -c '/etc/init.d/bcfg2-server start'
Starting Configuration Management Server: bcfg2-server     [  OK  ]
```

To verify that everything started ok, look for the running daemon and check the logs:

```
$ su -c 'tail /var/log/messages'
May 16 14:14:57 config01 bcfg2-server[2746]: service available at https://config01.local.ne
May 16 14:14:57 config01 bcfg2-server[2746]: serving bcfg2-server at https://config01.local
May 16 14:14:57 config01 bcfg2-server[2746]: serve_forever() [start]
May 16 14:14:57 config01 bcfg2-server[2746]: Handled 16 events in 0.009s
```

Run `bcfg2` to be sure you are able to communicate with the server:

```
$ su -c 'bcfg2 -vqne'

/usr/lib/python2.6/site-packages/Bcfg2/Client/Tools/rpmtools.py:23: DeprecationWarning: the
  import md5
Loaded plugins: presto, refresh-packagekit
Loaded tool drivers:
 Action     Chkconfig  POSIX      YUMng
Extra Package imsettings-libs 0.108.0-2.fc13.i686.
Extra Package PackageKit-device-rebind 0.6.4-1.fc13.i686.
...
Extra Package newt-python 0.52.11-2.fc13.i686.
Extra Package pulseaudio-gdm-hooks 0.9.21-6.fc13.i686.

Phase: initial
Correct entries:    0
Incorrect entries:  0
```

```
Total managed entries:      0
Unmanaged entries:  1314


Phase: final
Correct entries:    0
Incorrect entries:  0
Total managed entries:      0
Unmanaged entries:  1314
 Package:ConsoleKit                          Package:jasper-libs                         Pack
 Package:ConsoleKit-libs                     Package:java-1.5.0-gcj                      Pack
...
 Package:iw                                  Package:pcre                                Serv
 Package:jack-audio-connection-kit           Package:pcsc-lite                           Serv
```

The `bcfg2.conf` file contains only standard plugins so far.

```
$ su -c 'cat /etc/bcfg2.conf'

[server]
repository = /var/lib/bcfg2
plugins = Base,Bundler,Cfg,Metadata,Pkgmgr,Rules,SSHbase

[statistics]
sendmailpath = /usr/lib/sendmail
database_engine = sqlite3
# 'postgresql', 'mysql', 'mysql_old', 'sqlite3' or 'ado_mssql'.
database_name =
# Or path to database file if using sqlite3.
#<repository>/etc/brpt.sqlite is default path if left empty
database_user =
# Not used with sqlite3.
database_password =
# Not used with sqlite3.
database_host =
# Not used with sqlite3.
database_port =
# Set to empty string for default. Not used with sqlite3.
web_debug = True

[communication]
protocol = xmlrpc/ssl
password = test1234
certificate = /etc/bcfg2.crt
key = /etc/bcfg2.key
ca = /etc/bcfg2.crt

[components]
bcfg2 = https://config01.local.net:6789
```

Add the machines to Bcfg2

`bcfg2-admin` can be used to add a machine to Bcfg2 easily. You need to know the Fully Qualified Domain Name (FQDN) of ever system you want to control through Bcfg2.

```
bcfg2-admin client add <FQDN machine>
```

Bring your first machine under Bcfg2 control

Now it is time to get the first machine's configuration into the Bcfg2 repository. The server will be the first machine. It's already in the `Metadata/client.xml`.

Setup the Packages plugin First, replace **Pkgmgr** with **Packages** in the plugins line of `bcfg2.conf`. Then create *Packages/* directory in `/var/lib/bcfg2`

```
$ su -c 'mkdir /var/lib/bcfg2/Packages'
```

Create a `config.xml` file for the packages in `/var/lib/bcfg2/Packages` with the following content. Choose a mirror near your location according the Mirror list .

```
<Sources>
    <YUMSource>
    <Group>fedora-13</Group>
    <URL>ftp://fedora.tu-chemnitz.de/pub/linux/fedora/linux/releases/</URL>
    <Version>13</Version>
    <Component>Fedora</Component>
    <Arch>i386</Arch>
    <Arch>x86_64</Arch>
    </YUMSource>
</Sources>
```

Due to the Magic Groups, we need to modify our Metadata. Let's add a **fedora13** group which inherits a **fedora** group (this should replace the existing **redhat** group) present in `/var/lib/bcfg2/Metadata/groups.xml`. The resulting file should look something like this

```
<Groups version='3.0'>
    <Group profile='true' public='true' default='true' name='basic'>
       <Group name='fedora13'/>
    </Group>
    <Group name='fedora13'/>
       <Group name='fedora'/>
    <Group name='ubuntu'/>
    <Group name='debian'/>
    <Group name='freebsd'/>
    <Group name='gentoo'/>
    <Group name='fedora'/>
    <Group name='suse'/>
    <Group name='mandrake'/>
    <Group name='solaris'/>
 </Groups>
```

Note: When editing your xml files by hand, it is useful to occasionally run `bcfg2-repo-validate` to ensure that your xml validates properly.

Add a probe The next step for the client will be to have the proper arch group membership. For this, we will make use of the *server-plugins-grouping-dynamic_groups* capabilities of the Probes plugin. Add **Probes** to your plugins line in `bcfg2.conf` and create the Probe:

```
$ su -c 'mkdir /var/lib/bcfg2/Probes'
$ su -c 'cat /var/lib/bcfg2/Probes/groups'
#!/bin/sh

echo "group:`uname -m`"
```

Now a restart of `bcfg2-server` is needed:

```
$ su -c '/etc/init.d/bcfg2-server restart'
```

To test the Probe just run `bcfg2 -vqn`.

```
$ su -c 'bcfg2 -vqn'
Running probe group
Probe group has result:
group:i686
...
```

Start managing packages Add a base-packages bundle. Let's see what happens when we just populate it with the *yum* package. Create the `base-packages.xml` in your `Bundler/` directory with a entry for `yum`.

```
$ cat /var/lib/bcfg2/Bundler/base-packages.xml
<Bundle name='base-packages'>
        <Package name='yum'/>
</Bundle>
```

You need to reference the bundle from your `group.xml`. The resulting profile group might look something like this

```
<Group profile='true' public='true' default='true' name='basic'>
   <Bundle name='base-packages'/>
   <Group name='fedora13'/>
</Group>
```

Now if we run the client, we can see what this has done for us.:

```
output
```

As you can see, the Packages plugin has generated the dependencies required for the yum package automatically. The ultimate goal should be to move all the packages from the **Unmanaged** entries section to the **Managed** entries section. So, what exactly *are* those Unmanaged entries?:

```
output
```

Now you can go through these and continue adding the packages you want to your Bundle. After a while, I ended up with a minimal bundle that looks like this

```
<Bundle name='base-packages'>

</Bundle>
```

Now when I run the client, you can see I have only one unmanaged package:

```
outout
```

The gpg-pubkey packages are special in that they are not really packages. Currently, the way to manage them is using *BoundEntries*. So, after adding them, our Bundle now looks like this

Note: This does not actually control the contents of the files, you will need to do this part separately (see below).

```
<Bundle name='base-packages'>
        <BoundPackage name="gpg-pubkey" type="rpm">
                        <Instance simplefile="/etc/pki/rpm-gpg/RPM-GPG-KEY-CentOS-5" versi
                        <Instance simplefile="/etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL" version="
        </BoundPackage>
        <Package name='bcfg2-server'/>
        <Package name='exim'/>
        <Package name='grub'/>
        <Package name='kernel'/>
        <Package name='krb5-workstation'/>
        <Package name='m2crypto'/>
        <Package name='openssh-clients'/>
        <Package name='openssh-server'/>
        <Package name='prelink'/>
        <Package name='redhat-lsb'/>
        <Package name='rpm-build'/>
        <Package name='rsync'/>
        <Package name='sysklogd'/>
        <Package name='vim-enhanced'/>
        <Package name='yum'/>
</Bundle>
```

To actually push the gpg keys out via Bcfg2, you will need to manage the files as well. This can be done by adding Path entries for each of the gpg keys you want to manage

```
<Bundle name='base-packages'>
        <Path name='/etc/pki/rpm-gpg/RPM-GPG-KEY-CentOS-5'/>
        <Path name='/etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL'/>
        <BoundPackage name="gpg-pubkey" type="rpm">
                        <Instance simplefile="/etc/pki/rpm-gpg/RPM-GPG-KEY-CentOS-5" versi
                        <Instance simplefile="/etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL" version="
        </BoundPackage>
        <Package name='bcfg2-server'/>
        <Package name='exim'/>
        <Package name='grub'/>
        <Package name='kernel'/>
        <Package name='krb5-workstation'/>
        <Package name='m2crypto'/>
```

```
        <Package name='openssh-clients'/>
        <Package name='openssh-server'/>
        <Package name='prelink'/>
        <Package name='redhat-lsb'/>
        <Package name='rpm-build'/>
        <Package name='rsync'/>
        <Package name='sysklogd'/>
        <Package name='vim-enhanced'/>
        <Package name='yum'/>
</Bundle>
```

Then add the files to Cfg:

```
mkdir -p Cfg/etc/pki/rpm-gpg/RPM-GPG-KEY-CentOS-5
cp /etc/pki/rpm-gpg/RPM-GPG-KEY-CentOS-5 !$/RPM-GPG-KEY-CentOS-5
mkdir -p Cfg/etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL
cp /etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL !$/RPM-GPG-KEY-EPEL
```

Now, running the client shows only unmanaged Service entries. Woohoo!

Manage services Now let's clear up the unmanaged service entries by adding the following entries to our bundle...

```
<!-- basic services -->
<Service name='atd'/>
<Service name='avahi-daemon'/>
<Service name='bcfg2-server'/>
<Service name='crond'/>
<Service name='cups'/>
<Service name='gpm'/>
<Service name='lvm2-monitor'/>
<Service name='mcstrans'/>
<Service name='messagebus'/>
<Service name='netfs'/>
<Service name='network'/>
<Service name='postfix'/>
<Service name='rawdevices'/>
<Service name='sshd'/>
<Service name='syslog'/>
```

...and bind them in Rules

```
[root@centos ~]# cat /var/lib/bcfg2/Rules/services.xml
<Rules priority='1'>
        <!-- basic services -->
        <Service type='chkconfig' status='on' name='atd'/>
        <Service type='chkconfig' status='on' name='avahi-daemon'/>
        <Service type='chkconfig' status='on' name='bcfg2-server'/>
        <Service type='chkconfig' status='on' name='crond'/>
        <Service type='chkconfig' status='on' name='cups'/>
        <Service type='chkconfig' status='on' name='gpm'/>
        <Service type='chkconfig' status='on' name='lvm2-monitor'/>
        <Service type='chkconfig' status='on' name='mcstrans'/>
```

```
        <Service type='chkconfig' status='on' name='messagebus'/>
        <Service type='chkconfig' status='on' name='netfs'/>
        <Service type='chkconfig' status='on' name='network'/>
        <Service type='chkconfig' status='on' name='postfix'/>
        <Service type='chkconfig' status='on' name='rawdevices'/>
        <Service type='chkconfig' status='on' name='sshd'/>
        <Service type='chkconfig' status='on' name='syslog'/>
</Rules>
```

Now we run the client and see there are no more unmanaged entries!

```
$ su -c 'bcfg2 -veqn'
```

Adding Plugins

Git

Adding the *Git* plugins can preserve versioning information. The first step is to add *Git* to your plugin line:

```
plugins = Base,Bundler,Cfg,...,Git
```

For tracking the configuration files in the `/var/lib/bcfg2` directory a git repository need to be established:

```
git init
```

For more detail about the setup of git please refer to a git tutorial. The first commit can be the empty or the allready populated directory:

```
git add . && git commit -a
```

While running `bcfg2-info` the following line will show up:

```
Initialized git plugin with git directory = /var/lib/bcfg2/.git
```

11.7.5 Gentoo

This document tries to lay out anything Gentoo-specific that you need to know in order to use Bcfg2. Mostly that has to do with getting it to cooperate with the various pieces of Portage. Services, all things POSIX, and just about anything else that Bcfg2 does will work the same on Gentoo as on any other distribution. Bcfg2 is new on Gentoo; please let the list know if you find errors or omissions.

Installing Bcfg2

Early in July 2008, Bcfg2 was added to the Gentoo portage tree. So far it's only keyworded for ~x86, but we hope to see it soon in the amd64 and x64-solaris ports. If you're using Gentoo on some other architecture, it should still work provided that you have a reasonably up to date Python; try adding *app-admin/bcfg2 ~** to your */etc/portage/package.keywords* file.

If you don't use portage to install Bcfg2, you'll want to make sure you have all the prerequisites installed first. For a server, you'll need:

- `app-admin/gamin` or `app-admin/fam`
- `dev-python/lxml`

Clients will need at least:

- `app-portage/gentoolkit`

Package Repository

You'll need (to make) at least one archive of binary packages. The Portage driver calls `emerge` with the `-getbinpkgonly` option. See *make.conf(5)* and *emerge(1)* manpages, specifically the *PORTAGE_BINHOST* environment variable.

Time Saver: quickpkg

If you have a standing Gentoo machine that you want to preserve or propagate, you can generate a complete package archive based on the present state of the system by using the quickpkg utility. For example:

```
for pkg in `equery -q l` ; do quickpkg "=$pkg" ; done
```

...will leave you with a complete archive of all the packages on your system in `/usr/portage/packages/All`, which you can then move to your ftp server.

Cataloging Packages In Your Repository

Once you have a set of packages, you will need to create a catalog for them in `/var/lib/bcfg2/Pkgmgr`. Here's a template:

```
<PackageList uri='' type='portage' priority=''>
  <Group name=''>
    <Package name='' version=''/>
  </Group>
</PackageList>
```

...and a partially filled-out example, for our local Gentoo/VMware build:

```
<PackageList uri='ftp://filthy.uchicago.edu/200701-vmware/' type='portage' pricrity='0'>
  <Group name='gentoo-200701-vmware'>
    <Package name='app-admin/bcfg2' version='0.9.1_pre1'/>
    [...]
    <Package name='x11-wm/twm' version='1.0.1'/>
  </Group>
</PackageList>
```

The <*Group*> name (in our example, "gentoo-200701-vmware") should be included by any host which will draw its packages from this list. Our collection of packages for this class of machines is at the listed URI,

and we only have one collection of packages for this batch of machines so in our case the *priority* doesn't really matter, we've set it to *0*.

Notice that package name fields are in *CAT/TITLE* format.

Here is a hack which will generate a list of Package lines from a system's database of installed packages, especially useful in conjunction with the *quickpkg* example above:

```
#!/bin/bash
for pkg in `equery -q l` ; do
   title=`echo $pkg | sed -e 's/\(.*\)-\([0-9].*\)/\1/'`
   version=`echo $pkg | sed -e 's/\(.*\)-\([0-9].*\)/\2/'`
   echo "    <Package name='${title}' version='${version}'/>"
done
```

Configuring Client Machines

Set up `/etc/bcfg2.conf` the way you would for any other Bcfg2 client.

In `make.conf`, set *PORTAGE_BINHOST* to point to the URI of your package repository. You may want to create versions of `make.conf` for each package repository you maintain, with appropriate *PORTAGE_BINHOST* URI's in each, and associated with that package archive's group under `Cfg` – for example, we have `Cfg/etc/make.conf/make.conf.G99_gentoo-200701-vmware`. If a client host switches groups, and the new group needs a different set of packages, everything should just fall into place.

Pitfalls

Package Verification Issues

As of this writing (2007/01/31), we're aware of a number of packages marked stable in the Gentoo x86 tree which, for one reason or another, consistently fail to verify cleanly under *equery check*. In some cases (pam, openldap), files which don't (ever) exist on the system are nonetheless recorded in the package database; in some (python, Bcfg2, ahem), whole classes of files (.pyc and .pyo files) consistently fail their md5sum checks; and in others, the problem appears to be a discrepancy in the way that symlinks are created vs. the way they're recorded in the database. For example, in the OpenSSH package, /usr/bin/slogin is a symlink to ./ssh, but equery expects it to point to an unadorned ssh. An analogous situation exists with their manpages, leading to noise like this:

```
# equery check openssh
[ Checking net-misc/openssh-4.5_p1 ]
!!! /etc/ssh/sshd_config has incorrect md5sum
!!! /usr/bin/slogin does not point to ssh
!!! /usr/share/man/man1/slogin.1.gz does not point to ssh.1.gz
!!! /etc/ssh/ssh_config has incorrect md5sum
 * 62 out of 66 files good
```

We can ignore the lines for `ssh_config` and `sshd_config`; those will be caught by Bcfg2 as registered config files and handled appropriately.

Because Bcfg2 relies on the client system's native package reporting tool to judge the state of installed packages, complaints like these about trivial or intractable verification failures can trigger unnecessary bundle reinstalls when the Bcfg2 client runs. Bcfg2 will catch on after a pass or two that the situation isn't getting any better with repeated package installs, stop trying, and list those packages as "bad" in the client system's statistics.

Aside from filing bugs with the Gentoo package maintainers, your narrator has been unable to come up with a good approach to this. Maybe write a series of `Rules` definitions according to what the package database thinks it should find, and/or stage copies of affected files under `Cfg`, and associate those rules and files with the affected package in a bundle? Annoying but possibly necessary if you want your stats file to look good.

/boot

Gentoo as well as some other distros recommend leaving `/boot` unmounted during normal runtime. This can lead to trouble during verification and package installation, for example when `/boot/grub/grub.conf` turns up missing. The simplest way around this might just be to ensure that `/boot` is mounted whenever you run Bcfg2, possibly wrapping Bcfg2 in a script for the purpose. I've also thought about adding *Action* clauses to bundles for grub and our kernel packages, which would mount `/boot` before the bundle installs and unmount it afterward, but this doesn't get around the problem of those packages flunking verification.

11.7.6 NAT HOWTO

This page describes how to setup bcfg2 to properly function with a collection of clients behind NAT. It describes the issues, how the underlying portions of the bcfg2 system function, and how to correctly setup client metadata to cope with this environment.

Issues

Bcfg2, by default, uses ip address lookup to determine the identity of a client that has connected. This process doesn't work properly in the case of NATted hosts, because all requests from these clients come from the same external address when connecting to the server.

These client identification issues will manifest themselves in a number of ways:

- Inability to setup discrete clients with different profiles
- Incorrect sharing of probe results across clients in the same NAT pool
- Inability to bootstrap clients properly when client data is not predefined

Architectural Issues

Client identification is performed as the beginning of each client/server interaction. As of 0.9.3pre3, client identification via IP address can be completely short-circuited through the use of a client uuid. Setup of client uuids requires actions of both the client and server. On the server side, the client uuid must be added to the client record in Metadata/clients.xml. This attribute allows the server to use the user part of the

authentication to resolve the client's metadata. Also, either the location attribute should be set to floating, or the IP address of the NAT must be reflected in the address attribute. Once added, the Client entry in clients.xml will look like:

```
<Client profile="desktop" name="test1" pingable="N"
 uuid='9001ec29-1531-4b16-8198-a71bea093d0a' location='floating'/>
```

Alternatively, the Client entry can be setup like:

```
<Client profile="desktop" name="test1" pingable="N"
 uuid='9001ec29-1531-4b16-8198-a71bea093d0a' address='ip-address-of-NAT'/>
```

The difference between these definitions is explained in detail on the [wiki:Authentication] page, but in short, the second form requires that the client access the server from the NAT address, while the first form allows it to connect from any address provided it uses the proper uuid. (Client identification is orthogonal to the use of per-client passwords; this can be set in addition to the attributes above.)

Once this setup is done, each client must be configured to use the proper uuid upon any server interaction. This can be done using either the command line argument -u, or the setting "user" in the "communication" section of /etc/bcfg2.conf.

UUID Choice

When determining client UUIDs, one must take care to ensure that UUIDs are unique to the client. Any hardware-specific attribute, like a hash of a mac address would be sufficient. Alternatively, if a local hostname is unique, it may be used as well.

Automated Client Bootstrapping

Automated setup of new clients from behind NAT works by using the common password. For example:

```
/usr/sbin/bcfg2 -u ubik3 -p desktop -x <password>
```

It is not possible at this time to do automated setup without setting up a pre-shared per-client key.

11.7.7 Ubuntu

Note: This particular how to was done on lucid, but should apply to any other stable version of Ubuntu.

Install Bcfg2

We first need to install the server. For this example, we will use the bcfg2 server package from the bcfg2 PPA (note that there is also a version available in the ubuntu archives, but it is not as up to date).

Add the Ubuntu PPA listing to your APT sources

See http://trac.mcs.anl.gov/projects/bcfg2/wiki/PrecompiledPackages#UbuntuLucid

Install bcfg2-server

```
aptitude install bcfg2-server
```

Remove the default configuration preseeded by the ubuntu package:

```
root@lucid:~# rm -rf /etc/bcfg2* /var/lib/bcfg2
```

Initialize your repository

Now that you're done with the install, you need to intialize your repository and setup your bcfg2.conf. bcfg2-admin init is a tool which allows you to automate this process.:

```
root@lucid:~# bcfg2-admin init
Store bcfg2 configuration in [/etc/bcfg2.conf]:
Location of bcfg2 repository [/var/lib/bcfg2]:
Input password used for communication verification (without echoing; leave blank for a ran
What is the server's hostname: [lucid]
Input the server location [https://lucid:6789]:
Input base Operating System for clients:
1: Redhat/Fedora/RHEL/RHAS/Centos
2: SUSE/SLES
3: Mandrake
4: Debian
5: Ubuntu
6: Gentoo
7: FreeBSD
: 5
Generating a 1024 bit RSA private key
.....................................................................................+++
...++++++
writing new private key to '/etc/bcfg2.key'
-----
Signature ok
subject=/C=US/ST=Illinois/L=Argonne/CN=lucid
Getting Private key
Repository created successfuly in /var/lib/bcfg2
```

Of course, change responses as necessary.

Start the server

You are now ready to start your bcfg2 server for the first time.:

```
root@lucid:~# /etc/init.d/bcfg2-server start
root@lucid:~# tail /var/log/syslog
Dec 17 22:07:02 lucid bcfg2-server[17523]: serving bcfg2-server at https://lucid:6789
Dec 17 22:07:02 lucid bcfg2-server[17523]: serve_forever() [start]
Dec 17 22:07:02 lucid bcfg2-server[17523]: Processed 16 fam events in 0.502 seconds. 0 coal
```

Run bcfg2 to be sure you are able to communicate with the server:

```
root@lucid:~# bcfg2 -vqn
Loaded tool drivers:
 APT      Action   DebInit  POSIX

Phase: initial
Correct entries:        0
Incorrect entries:      0
Total managed entries:  0
Unmanaged entries:      382


Phase: final
Correct entries:        0
Incorrect entries:      0
Total managed entries:  0
Unmanaged entries:      382
```

Bring your first machine under Bcfg2 control

Now it is time to get your first machine's configuration into your Bcfg2 repository. Let's start with the server itself.

Setup the Packages plugin

Replace Pkgmgr with Packages in the plugins line of `bcfg2.conf`:

```
root@lucid:~# cat /etc/bcfg2.conf
[server]
repository = /var/lib/bcfg2
plugins = Base,Bundler,Cfg,Metadata,Packages,Rules,SSHbase

[statistics]
sendmailpath = /usr/lib/sendmail
database_engine = sqlite3
# 'postgresql', 'mysql', 'mysql_old', 'sqlite3' or 'ado_mssql'.
database_name =
# Or path to database file if using sqlite3.
#<repository>/etc/brpt.sqlite is default path if left empty
database_user =
# Not used with sqlite3.
database_password =
# Not used with sqlite3.
database_host =
# Not used with sqlite3.
database_port =
# Set to empty string for default. Not used with sqlite3.
web_debug = True

[communication]
protocol = xmlrpc/ssl
```

```
password = secret
certificate = /etc/bcfg2.crt
key = /etc/bcfg2.key
ca = /etc/bcfg2.crt

[components]
bcfg2 = https://lucid:6789
```

Create Packages layout (as per *Example usage*) in `/var/lib/bcfg2`

```
root@lucid:~# mkdir /var/lib/bcfg2/Packages
root@lucid:~# cat /var/lib/bcfg2/Packages/config.xml
<Sources>
  <APTSource>
    <Group>ubuntu-lucid</Group>
    <URL>http://us.archive.ubuntu.com/ubuntu</URL>
    <Version>lucid</Version>
    <Component>main</Component>
    <Component>multiverse</Component>
    <Component>restricted</Component>
    <Component>universe</Component>
    <Arch>amd64</Arch>
    <Arch>i386</Arch>
  </APTSource>
</Sources>
```

Due to the Magic Groups, we need to modify our Metadata. Let's add an **ubuntu-lucid** group which inherits the **ubuntu** group already present in `/var/lib/bcfg2/Metadata/groups.xml`. The resulting file should look something like this

```
<Groups version='3.0'>
   <Group profile='true' public='true' default='true' name='basic'>
      <Group name='ubuntu-lucid'/>
   </Group>
   <Group name='ubuntu-lucid'>
      <Group name='ubuntu'/>
   </Group>
   <Group name='ubuntu'/>
   <Group name='debian'/>
   <Group name='freebsd'/>
   <Group name='gentoo'/>
   <Group name='redhat'/>
   <Group name='suse'/>
   <Group name='mandrake'/>
   <Group name='solaris'/>
</Groups>
```

Note: When editing your xml files by hand, it is useful to occasionally run *bcfg2-repo-validate* to ensure that your xml validates properly.

The last thing we need is for the client to have the proper arch group membership. For this, we will make use of the *server-plugins-grouping-dynamic_groups* capabilities of the Probes plugin. Add Probes to your plugins line in `bcfg2.conf` and create the Probe.

```
root@lucid:~# grep plugins /etc/bcfg2.conf
plugins = Base,Bundler,Cfg,Metadata,Packages,Probes,Rules,SSHbase
root@lucid:~# mkdir /var/lib/bcfg2/Probes
root@lucid:~# cat /var/lib/bcfg2/Probes/groups
#!/bin/sh

ARCH=`uname -m`
case "$ARCH" in
    "x86_64")
        echo "group:amd64"
    ;;
    "i686")
        echo "group:i386"
    ;;
esac
```

Now we restart the bcfg2-server:

```
root@lucid:~# /etc/init.d/bcfg2-server restart
Stopping Configuration Management Server:  * bcfg2-server
Starting Configuration Management Server:  * bcfg2-server
root@lucid:~# tail /var/log/syslog
Dec 17 22:36:47 lucid bcfg2-server[17937]: Packages: File read failed; falling back to file
Dec 17 22:36:47 lucid bcfg2-server[17937]: Packages: Updating http://us.archive.ubuntu.com/
Dec 17 22:36:54 lucid bcfg2-server[17937]: Packages: Updating http://us.archive.ubuntu.com/
Dec 17 22:36:55 lucid bcfg2-server[17937]: Packages: Updating http://us.archive.ubuntu.com/
Dec 17 22:36:56 lucid bcfg2-server[17937]: Packages: Updating http://us.archive.ubuntu.com/
Dec 17 22:37:27 lucid bcfg2-server[17937]: Failed to read file probed.xml
Dec 17 22:37:27 lucid bcfg2-server[17937]: Loading experimental plugin(s): Packages
Dec 17 22:37:27 lucid bcfg2-server[17937]: NOTE: Interfaces subject to change
Dec 17 22:37:27 lucid bcfg2-server[17937]: service available at https://lucid:6789
Dec 17 22:37:27 lucid bcfg2-server[17937]: serving bcfg2-server at https://lucid:6789
Dec 17 22:37:27 lucid bcfg2-server[17937]: serve_forever() [start]
Dec 17 22:37:28 lucid bcfg2-server[17937]: Processed 17 fam events in 0.502 seconds. 0 coal
```

Start managing packages

Add a base-packages bundle. Let's see what happens when we just populate it with the ubuntu-standard package.

```
root@lucid:~# cat /var/lib/bcfg2/Bundler/base-packages.xml
<Bundle name='base-packages'>
        <Package name='ubuntu-standard'/>
</Bundle>
```

You need to reference the bundle from your Metadata. The resulting profile group might look something like this

```
<Group profile='true' public='true' default='true' name='basic'>
   <Bundle name='base-packages'/>
   <Group name='ubuntu-lucid'/>
</Group>
```

Now if we run the client in debug mode (-d), we can see what this has done for us.:

```
root@lucid:~# bcfg2 -vqdn
Running probe groups
Probe groups has result:
amd64
Loaded tool drivers:
 APT      Action   DebInit  POSIX
The following packages are specified in bcfg2:
 ubuntu-standard
The following packages are prereqs added by Packages:
 adduser             debconf             hdparm              libdevmapper1.02.1  libk5crypt
 apt                 debianutils         info                libdns53            libkeyuti
 aptitude            dmidecode           install-info        libelf1             libkrb5-3
 at                  dnsutils            iptables            libept0             libkrb5sup
 base-files          dosfstools          libacl1             libgcc1             liblwres50
 base-passwd         dpkg                libattr1            libgdbm3            libmagic1
 bsdmainutils        ed                  libbind9-50         libgeoip1           libmpfr1lo
 bsdutils            file                libc-bin            libgmp3c2           libncurses
 cpio                findutils           libc6               libgssapi-krb5-2    libncurses
 cpp                 ftp                 libcap2             libisc50            libpam-mod
 cpp-4.4             gcc-4.4-base        libcomerr2          libisccc50          libpam-run
 cron                groff-base          libcwidget3         libisccfg50         libpam0g

Phase: initial
Correct entries:        101
Incorrect entries:      0
Total managed entries:  101
Unmanaged entries:      281


Phase: final
Correct entries:        101
Incorrect entries:      0
Total managed entries:  101
Unmanaged entries:      281
```

As you can see, the Packages plugin has generated the dependencies required for the ubuntu-standard package for us automatically. The ultimate goal should be to move all the packages from the **Unmanaged** entries section to the **Managed** entries section. So, what exactly *are* those Unmanaged entries?:

```
root@lucid:~# bcfg2 -vqen
Running probe groups
Probe groups has result:
amd64
Loaded tool drivers:
 APT      Action   DebInit  POSIX

Phase: initial
Correct entries:        101
Incorrect entries:      0
Total managed entries:  101
Unmanaged entries:      281
```

```
Phase: final
Correct entries:        101
Incorrect entries:      0
Total managed entries:  101
Unmanaged entries:      281
 Package:apparmor
 Package:apparmor-utils
 Package:apport
 ...
```

Now you can go through these and continue adding the packages you want to your Bundle. Note that `aptitude why` is useful when trying to figure out the reason for a package being installed. Also, deborphan is helpful for removing leftover dependencies which are no longer needed. After a while, I ended up with a minimal bundle that looks like this

```
<Bundle name='base-packages'>
        <Package name='bash-completion'/>
        <Package name='bcfg2-server'/>
        <Package name='debconf-i18n'/>
        <Package name='deborphan'/>
        <Package name='diffutils'/>
        <Package name='e2fsprogs'/>
        <Package name='fam'/>
        <Package name='grep'/>
        <Package name='grub-pc'/>
        <Package name='gzip'/>
        <Package name='hostname'/>
        <Package name='krb5-config'/>
        <Package name='krb5-user'/>
        <Package name='language-pack-en-base'/>
        <Package name='linux-generic'/>
        <Package name='linux-headers-generic'/>
        <Package name='login'/>
        <Package name='manpages'/>
        <Package name='mlocate'/>
        <Package name='ncurses-base'/>
        <Package name='openssh-server'/>
        <Package name='python-fam'/>
        <Package name='tar'/>
        <Package name='ubuntu-minimal'/>
        <Package name='ubuntu-standard'/>
        <Package name='vim'/>
        <Package name='vim-runtime'/>

        <!-- PreDepends -->
        <Package name='dash'/>
        <Package name='initscripts'/>
        <Package name='libdbus-1-3'/>
        <Package name='libnih-dbus1'/>
        <Package name='lzma'/>
        <Package name='mountall'/>
        <Package name='sysvinit-utils'/>
        <Package name='sysv-rc'/>
```

```
        <!-- vim dependencies -->
        <Package name='libgpm2'/>
        <Package name='libpython2.6'/>
</Bundle>
```

As you can see below, I no longer have any unmanaged packages.

```
root@lucid:~# bcfg2 -vqen
Running probe groups
Probe groups has result:
amd64
Loaded tool drivers:
 APT      Action   DebInit  POSIX

Phase: initial
Correct entries:        247
Incorrect entries:      0
Total managed entries:  247
Unmanaged entries:      10


Phase: final
Correct entries:        247
Incorrect entries:      0
Total managed entries:  247
Unmanaged entries:      10
 Service:bcfg2         Service:fam           Service:killprocs     Service:rc.local      Se
 Service:bcfg2-server  Service:grub-common   Service:ondemand      Service:rsync         Se
```

Manage services

Now let's clear up the unmanaged service entries by adding the following entries to our bundle...

```
<!-- basic services -->
<Service name='bcfg2'/>
<Service name='bcfg2-server'/>
<Service name='fam'/>
<Service name='grub-common'/>
<Service name='killprocs'/>
<Service name='ondemand'/>
<Service name='rc.local'/>
<Service name='rsync'/>
<Service name='single'/>
<Service name='ssh'/>
```

...and bind them in Rules

```
root@lucid:~# cat /var/lib/bcfg2/Rules/services.xml
<Rules priority='1'>
        <!-- basic services -->
        <Service type='deb' status='on' name='bcfg2'/>
        <Service type='deb' status='on' name='bcfg2-server'/>
```

```
        <Service type='deb' status='on' name='fam'/>
        <Service type='deb' status='on' name='grub-common'/>
        <Service type='deb' status='on' name='killprocs'/>
        <Service type='deb' status='on' name='ondemand'/>
        <Service type='deb' status='on' name='rc.local'/>
        <Service type='deb' status='on' name='rsync'/>
        <Service type='deb' status='on' name='single'/>
        <Service type='deb' status='on' name='ssh'/>
</Rules>
```

Now we run the client and see there are no more unmanaged entries!

```
root@lucid:~# bcfg2 -vqn
Running probe groups
Probe groups has result:
amd64
Loaded tool drivers:
 APT      Action   DebInit  POSIX

Phase: initial
Correct entries:        257
Incorrect entries:      0
Total managed entries:  257
Unmanaged entries:      0

All entries correct.

Phase: final
Correct entries:        257
Incorrect entries:      0
Total managed entries:  257
Unmanaged entries:      0

All entries correct.
```

Dynamic (web) reports

See installation instructions at *server-reports-install*

11.7.8 Using bcfg2-info

`bcfg2-info` is a tool for introspecting server functions. It is useful for understanding how the server is interpreting your repository. It consists of the same logic executed by the server to process the repository and produce configuration specifications, just without all of the network communication code. Think of `bcfg2-info` as `bcfg2-server` on a stick. It is a useful location to do testing and staging of new configuration rules, prior to deployment. This is particularly useful when developing templates, or developing Bcfg2 plugins.

Getting Started

First, fire up the bcfg2-info interpreter.

```
[0:464] bcfg2-info
Loading experimental plugin(s): Packages
NOTE: Interfaces subject to change
Handled 8 events in 0.006s
Handled 4 events in 0.035s
Welcome to bcfg2-info
Type "help" for more information
>
```

At this point, the server core has been loaded up, all plugins have been loaded, and the `bcfg2-info` has both read the initial state of the Bcfg2 repository, as well as begun monitoring it for changes. Like *bcfg2-server*, `bcfg2-info` monitors the repository for changes, however, unlike *bcfg2-server*, it does not process change events automatically. File modification events can be processed by explicitly calling the **update** command. This will process the events, displaying the number of events processed and the amount of time taken by this processing. If no events are available, no message will be displayed. For example, after a change to a file in the repository:

```
>update
Handled 1 events in 0.001s
> update
>
```

This explicit update process allows you to control the update process, as well as see the precise changes caused by repository modifications.

`bcfg2-info` has several builtin commands that display the state of various internal server core state. These are most useful for examining the state of client metadata, either for a single client, or for clients overall.

clients displays a list of clients, along with their profile groups

groups displays a list of groups, the inheritance hierarchy, profile status, and category name, if there is one.

showclient displays full metadata information for a client, including profile group, group memberships, bundle list, and any connector data, like Probe values or Property info.

Debugging Configuration Rules

In addition to the commands listed above for viewing client metadata, there are also commands which can shed light on the configuration generation process. Recall that configuration generation occurs in three major steps:

1. Resolve client metadata
2. Build list of entries for the configuration
3. Bind host-specific version of each entry

Step *1* can be viewed with the commands presented in the previous section. The latter two steps can be examined using the following commands.

showentries displays a list of entries (optionally filtered by type) that appear in a client's configuration specification

buildfile Perform the entry binding process on a single entry, displaying its results. This command is very useful when developing configuration file templates.

build Build the full configuration specification and write it to a file.

mappings displays the entries handled by the plugins loaded by the server core. This command is useful when the server reports a bind failure for an entry.

Debugging and Developing Bcfg2

`bcfg2-info` loads a full Bcfg2 server core, so it provides the ideal environment for developing and debugging Bcfg2. Because it is hard to automate this sort of process, we have only implemented two commands in `bcfg2-info` to aid in the process.

profile The profile command produces python profiling information for other `bcfg2-info` commands. This can be used to track performance problems in configuration generation.

debug The debug command exits the `bcfg2-info` interpreter loop and drops to a python interpreter prompt. The Bcfg2 server core is available in this namespace as "self". Full documentation for the server core is out of scope for this document. This capability is most useful to call into plugin methods, often with setup calls or the enabling of diagnostics.

It is possible to return to the `bcfg2-info` command loop by exiting the python interpreter with ^D.

There is built-in support for IPython in `bcfg2-info`. If IPython is installed, dropping into debug mode in `bcfg2-info` will use the IPython interpreter by default.

11.7.9 Using Bcfg2 With CentOS

This section covers specific topics for using Bcfg2 with CentOS. Most likely the tips on this page also apply to other members of the Red Hat family of Linux operating systems.

From Source

Install Prerequisities

While you can go about building all these things from source, this how to will try and meet the dependencies using packages from EPEL or RPMforge. The *el5* package should be compatible with CentOS 5.x.

EPEL:

```
[root@centos ~]# rpm -Uvh http://download.fedora.redhat.com/pub/epel/5/i386/epel-release-5-
```

RPMforge:

```
[root@centos ~]# rpm -Uvh http://dag.wieers.com/rpm/packages/rpmforge-release/rpmforge-rele
```

Note: Be careful with mixing package repositories.

Now you can install the rest of the prerequisites:

```
[root@centos ~]# yum install python-genshi python-cheetah python-lxml
```

Build Packages from source

- After installing subversion, check out a copy of trunk

  ```
  [root@centos redhat]# svn co https://svn.mcs.anl.gov/repos/bcfg/trunk/bcfg2
  ```

- Install the `fedora-packager` package

  ```
  [root@centos ~]# yum install fedora-packager
  ```

- A directory structure for the RPM build process has to be established.

  ```
  [you@centos ~]$ rpmdev-setuptree
  ```

- Change to the *redhat* directory of the checked out Bcfg2 source:

  ```
  [you@centos ~]$ cd bcfg2/redhat/
  ```

- In the particular directory is a Makefile which will do the job of building the RPM packages. You can do this as root, but it's not recommanded:

  ```
  [you@centos redhat]$ make
  ```

- Now the new RPM package can be installed. Please adjust the path to your RPM package:

  ```
  [root@centos ~]# rpm -ihv /home/YOU/rpmbuild/RPMS/noarch/bcfg2-server-1.0.0-0.2r583
  ```

Install Packages from Package repository

To install the bcfg2-server and bcfg2 from a package repository, just use Yum to do it:

```
[root@centos ~]# yum install bcfg2-server bcfg2
```

11.7.10 Version control systems

The sections in this guide do only cover the basics steps in the setup of the different version control system for the usage with the Bcfg2 plugin support. More more details about

Git

Adding the *Git* plugins can preserve versioning information. The first step is to add **Git** to your plugin line:

```
plugins = Base,Bundler,Cfg,...,Git
```

For tracking the configuration files in the `/var/lib/bcfg2` directory a git repository need to be established:

```
git init
```

For more detail about the setup of git please refer to a git tutorial. The first commit can be the empty or the already populated directory:

```
git add . && git commit -a
```

While running `bcfg2-info` the following line will show up:

```
Initialized git plugin with git directory = /var/lib/bcfg2/.git
```

Mercurial

For the *Mercurial (Hg)* plugin are the same changes needed as for git.

```
plugins = Base,Bundler,Cfg,...,Mercurial
```

The repository must be initialized:

```
hg init
```

Mercurial will not commit the files to the repository until a user name is defined in `/var/lib/bcfg2/.hg/`

```
cat <<END_ENTRY >> /var/lib/bcfg2/.hg/hgrc
[ui]
username = Yor name <you@example.com>
END_ENTRY
```

Now you are able to make submissions to the repository:

```
hg commit
```

While running `bcfg2-info` the following line will show up:

```
Initialized hg plugin with hg directory = /var/lib/bcfg2/.hg
```

Darcs

If you wish to use the *Darcs* plugin an entry has to be made in the `bcfg2.conf` file.:

```
plugins = Base,Bundler,Cfg,...,Darcs
```

The dracs repository must be initialized:

```
darcs initialize
```

To commit to the darcs repository an author must be added to the `_darcs/prefs/author` file. If the `author` file is missing, darcs will ask you to enter your e-mail address.

```
cat <<END_ENTRY >> /var/lib/bcfg2/_darcs/prefs/author
you@example.com
END_ENTRY
```

All files in the `/var/lib/bcfg2` should be added to darcs now:

```
darcs add *
```

After that you can submit them to the repository:

```
darcs record
```

While running `bcfg2-info` the following line will show up:

```
Initialized Darcs plugin with darcs directory = /var/lib/bcfg2/_darcs
```

Cvs

If you wish to use the *Darcs* plugin an entry has to be made in the `bcfg2.conf` file.:

```
plugins = Base,Bundler,Cfg,...,Cvs
```

The CVS repository must be initialized:

```
cvs -d /var/lib/bcfg2 init
```

11.7.11 Dynamic (web) Reports installation

The first step is to install the needed software components like the Django framework and the database (SQlite2). All packages for Fedora are in the Fedora Package Collection or in EPEL for CentOS/RHEL:

```
[root@system01 ~]# yum -y install Django python-simplejson python-sqlite2
```

Of course is a web server needed as well:

```
[root@system01 ~]# yum -y install httpd mod_python
```

The same packages are needed for Ubuntu systems:

```
[root@system01 ~]# aptitude install python-django apache2 libapache2-mod-python
```

Now we need to create the sqlite database. Use the following command on Fedora, CentOS, or RHEL.:

```
[root@system01 ~]# python /usr/lib/python2.4/site-packages/Bcfg2/Server/Reports/manage.py s
Creating table auth_permission
Creating table auth_group
Creating table auth_user
```

```
Creating table auth_message
Creating table django_content_type
Creating table django_session
Creating table django_site
Creating table django_admin_log
Creating table reports_client
Creating table reports_ping
Creating table reports_interaction
Creating table reports_reason
Creating table reports_entries
Creating table reports_entries_interactions
Creating table reports_performance
Creating table reports_internaldatabaseversion

You just installed Django's auth system, which means you don't have any superusers defined
Would you like to create one now? (yes/no): no
Installing index for auth.Permission model
Installing index for auth.Message model
Installing index for admin.LogEntry model
Installing index for reports.Client model
Installing index for reports.Ping model
Installing index for reports.Interaction model
Installing index for reports.Entries model
Installing index for reports.Entries_interactions model
```

Note: There are different versions of Python available. If you are unsure about your installed version use the following line instead of the line above.:

```
[root@system01 ~]# PYVER=`python -c 'import sys;print(sys.version[0:3])'`; python /usr/lib/
```

The path on Ubuntu systems is different. Please use the same path as shown in the following command to execute the script on an Ubuntu machine in the next steps:

```
[root@system01 ~]# python /usr/share/pyshared/Bcfg2/Server/Reports/manage.py syncdb
Creating table auth_permission
Creating table auth_group
Creating table auth_user
Creating table auth_message
Creating table django_content_type
Creating table django_session
Creating table django_site
Creating table django_admin_log
Creating table reports_client
Creating table reports_ping
Creating table reports_interaction
Creating table reports_reason
Creating table reports_entries
Creating table reports_entries_interactions
Creating table reports_performance
Creating table reports_internaldatabaseversion

You just installed Django's auth system, which means you don't have any superusers defined
Would you like to create one now? (yes/no): no
```

```
Installing index for auth.Permission model
Installing index for auth.Message model
Installing index for admin.LogEntry model
Installing index for reports.Client model
Installing index for reports.Ping model
Installing index for reports.Interaction model
Installing index for reports.Entries model
Installing index for reports.Entries_interactions model
```

The server should be tested to make sure that there are no mistakes:

```
[root@system01 ~]# python /usr/lib/python2.6/site-packages/Bcfg2/Server/Reports/manage.py t
Creating test database...
Creating table auth_permission
Creating table auth_group
Creating table auth_user
Creating table auth_message
Creating table django_content_type
Creating table django_session
Creating table django_site
Creating table django_admin_log
Creating table reports_client
Creating table reports_ping
Creating table reports_interaction
Creating table reports_reason
Creating table reports_entries
Creating table reports_entries_interactions
Creating table reports_performance
Creating table reports_internaldatabaseversion
Installing index for auth.Permission model
Installing index for auth.Message model
Installing index for admin.LogEntry model
Installing index for reports.Client model
Installing index for reports.Ping model
Installing index for reports.Interaction model
Installing index for reports.Entries model
Installing index for reports.Entries_interactions model
Validating models...
0 errors found

Django version 1.1.1, using settings 'Reports.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Add DBStats to the plugins line of `bcfg2.conf`. The resulting **[server]** section should look something like this:

```
[server]
repository = /var/lib/bcfg2
plugins = Base,Bundler,Cfg,DBStats,Metadata,Packages,Probes,Rules,SSHbase
```

Start/restart the Bcfg2 server:

```
[root@system01 ~]# /etc/init.d/bcfg2-server restart
```

Run the Bcfg2 client in order to populate the statistics database (this run should take a bit longer since you are uploading the client statistics to the database).

Download the static reports content:

```
[root@system01 ~]# cd /var/www/
[root@system01 ~]# svn co https://svn.mcs.anl.gov/repos/bcfg/trunk/bcfg2/reports
```

Configure Apache using *Apache configuration for web-based reports* as a guide

Copy server/statistics sections of `bcfg2.conf` to `/etc/bcfg2-web.conf` (make sure it is world-readable). You should then have something like this:

```
[server]
repository = /var/lib/bcfg2
plugins = Base,Bundler,Cfg,DBStats,Metadata,Packages,Probes,Rules,SSHbase

[statistics]
sendmailpath = /usr/lib/sendmail
database_engine = sqlite3
# 'postgresql', 'mysql', 'mysql_old', 'sqlite3' or 'ado_mssql'.
database_name =
# Or path to database file if using sqlite3.
#<repository>/etc/brpt.sqlite is default path if left empty
database_user =
# Not used with sqlite3.
database_password =
# Not used with sqlite3.
database_host =
# Not used with sqlite3.
database_port =
# Set to empty string for default. Not used with sqlite3.
web_debug = True
```

Restart apache and point a browser to your Bcfg2 server.

If using sqlite be sure the sql database file and directory containing the database are writable to apache.

11.8 Tools

In the `tools/` directory are several tools collected. Those tools can help you to maintain your Bcfg2 configuration, to make the initial setup easier, or to do some other tasks.

http://trac.mcs.anl.gov/projects/bcfg2/browser/trunk/bcfg2/tools

CHAPTER

TWELVE

UNSORTED DOCS

These docs have yet to be sorted properly. The content for them can be found at the TitleIndex page on Trac. Most should be converted to sphinx. Some may not need any conversion (e.g. The Download page). Once converted and put in the proper place, you can remove the item from the list below.

- *Plugins/Snapshots*
- *PrecompiledPackages*
- *Publications*
- *QuickStart2*
- *QuickStart3*
- *SchemaEvolution*
- *SecurityDevPlan*
- *ServerSideOverview*
- *UpgradeTesting*
- *VimSnippetSupport*

12.1 Ways to get help

12.1.1 Interactive Help

- [wiki:IRCChannel IRC Channel, with indexed archives]
- [wiki:MailingList Mailing list, with indexed archives]

Note that the IRC channel tends to be much busier than the mailing list; use whichever seems most appropriate for your query, but don't let the lack of mailing list activity make you think the project isn't active.

12.1.2 Frequently Asked Questions

- [wiki:FAQ The FAQ]

12.1.3 Examples

- There are examples sprinkled throughout this wiki; we should link to them from here.
- The [http://www.fsf.org Free Software Foundation] is (very slowly) working towards having configurations for the majority of the machines it administers available via [http://config.fsf.org config.fsf.org]. This is a tie-in with the [http://autonomo.us/2008/07/franklin-street-statement/ Franklin Street Statement on Freedom and Network Services] (FSF offices are on Franklin Street). Documentation on how to have a public access Bcfg2 configuration repository will be at PublicRepository.

12.1.4 Manuals

- The current canonical source of documentation are pages on this wiki ([wiki:UsingBcfg2]). Please mail the MailingList for editor access to this wiki.
- There is a printed manual in the SAGE short topics series, "!#19: Configuration Management with Bcfg2", that you can [https://db.usenix.org/cgi-bin/sage/booklets/order.cgi order] for $20 (or get for free if you are a [http://www.sage.org/index.html SAGE] member and haven't gotten a booklet yet during your current membership year). The book includes documentation up to and including most features in Bcfg2 0.9.6. Note that all proceeds from the sale of this book go to SAGE.

12.1.5 FLOSS Manual Project

A project is getting started to make a user-contributed manual using the [http://en.flossmanuals.net/ FLOSS Manuals] web site and tools. The intention is for this manual to be based on but not a verbatim copy of the information on the wiki, formatted in a way that is easier for new users to read, and written mostly by users of Bcfg2, rather than the authors of Bcfg2. This manual will also be free (as in freedom and price).

One important point is that new contributors can get edit access to the manual in about a minute, and the manual is edited via WYSIWYG tools, so there should be pretty much no barrier for new manual authors to get started.

There will be an announcement to the mailing list about this soon.

If you are seriously interested in dedicating time to this manual, it would make sense to read the [http://en.flossmanuals.net/FLOSSManuals FLOSS Manuals Manual] (free online) and the [https://db.usenix.org/cgi-bin/sage/booklets/order.cgi Configuration Management with Bcfg2] manual ($20). If you are willing to commit time to manual writing, would like physical copies of these manuals, and purchasing them would be a financial hardship for you, email [http://pobox.com/~dclark Danny Clark] at dclark@pobox.com (ping djbclark on [wiki:IRCChannel #bcfg2 irc] if you don't get a reply) with your postal address (don't be shy, I already bought a bunch of these, and they aren't doing much good sitting on my shelf :-).

You can get to the Bcfg2 FLOSS Manual at http://docs.bcfg2.org (which just redirects to http://en.flossmanuals.net/bin/view/BCFG2).

12.2 HOWTOs

Here are several howtos that describe different aspects of Bcfg2 deployment

- *authentication* - a description of the Bcfg2 authentication infrastructure
- AnnotatedExamples - a description of basic Bcfg2 specification operations
- BuildingDebianPackages - How to build debian packages
- *unsorted-gentoo* - Issues specific to running Bcfg2 on Gentoo
- *TCheetah* - Howto use the TCheetah template plugin
- *Hostbase* - How to use the Hostbase plugin and web interface
- *Probes* - How to use Probes to gather information from a client machine.
- *Actions* - How to use Actions
- *unsorted-dynamic_groups* - Using dynamic groups
- *Paranoid mode* - How to run an update in paranoid mode

12.3 Python SSL

The ssl module can be found here.

With this change, SSL certificate based client authentication is supported. In order to use this, based CA-type capabilities are required. A central CA needs to be created, with each server and all clients getting a signed cert. See [wiki:Authentication] for details.

Setting up keys is accomplished with three settings, each in the "*[communication]*" section of `bcfg2.conf`:

```
key = /path/to/ssl private key
certificate = /path/to/signed cert for that key
ca = /path/to/cacert.pem
```

12.3.1 Python SSL Backport Packaging

Both the Bcfg2 server and client are able to use the in-tree ssl module included with python 2.6. The client is also able to still use M2Crypto. A python ssl backport exists for 2.3, 2.4, and 2.5. With this, M2Crypto is not needed, and tlslite is no longer included with Bcfg2 sources. See [wiki:Authentication] for details.

To build a package of the ssl backport for .deb based distributions that don't ship with python 2.6, you can follow these instructions, which use stdeb. Alternatively if you happen to have .deb packaging skills, it would be great to get policy-complaint .debs into the major deb-based distributions.

The following commands were used to generate `this` debian package The `easy_install` command can be found in the *python-setuptools* package.:

```
sudo aptitude install python-all-dev fakeroot
sudo easy_install stdeb
wget http://pypi.python.org/packages/source/s/ssl/ssl-1.14.tar.gz#md5=4e08aae0cd2c7388d1b4
tar xvfz ssl-1.14.tar.gz
cd ssl-1.14
stdeb_run_setup
cd deb_dist/ssl-1.14
dpkg-buildpackage -rfakeroot -uc -us
sudo dpkg -i ../python-ssl_1.14-1_amd64.deb
```

Note: Version numbers for the SSL module have changed.

For complete Bcfg2 goodness, you'll also want to package stdeb using stdeb. The completed debian package can be grabbed from `here`, which was generated using the following:

```
sudo aptitude install apt-file
wget http://pypi.python.org/packages/source/s/stdeb/stdeb-0.3.tar.gz#md5=e692f745597dcdd93
tar xvfz stdeb-0.3.tar.gz
cd stdeb-0.3
stdeb_run_setup
cd deb_dist/stdeb-0.3
dpkg-buildpackage -rfakeroot -uc -us
sudo dpkg -i ../python-stdeb_0.3-1_all.deb
```

12.4 Notes on possible Windows support

- Windows Management Instrumentation (WMI) should be used wherever possible; there is an excellent [http://tgolden.sc.sabren.com/python/wmi.html WMI Python Module] available, which also comes with a [http://tgolden.sc.sabren.com/python/wmi_cookbook.html WMI Cookbook].
- Before Windows 2003 SP1, on 64-bit machines there are [http://msdn2.microsoft.com/en-us/library/aa393067.aspx no API or WMI calls] to get to many 32-bit windows functions (such as the 32-bit registry) from 64-bit programs, and vice versa. There also is no (official) x86_64 native python distributions for Windows pre-Python 2.5. So the choice would be:

1. Only support Windows in Python 2.5+ (which wouldn't be that bad because part of the build process would probably be to create stand-alone bcfg2 executables using [http://www.py2exe.org/ py2exe]). For 64-bit support there would have to be some kind of convoluted py2exe build process that built some things with 32-bit python and some things with 64-bit python.
2. Wrap external command-line programs such as winreg, which is part of [http://dmst.aueb.gr/dds/sw/outwit/ outwit], and screen scrape. Each external command-line program would need to be compiled into 32 and 64 bit versions. This approach might lead to licensing annoyances and having binary blobs in source control.

12.4.1 Services

With the exception of 32/64 bit issues, Windows Services support should be pretty trivial; it would differ from *nix services in that it would be done via WMI API calls and not a 3rd party python module or

wrapping a binary.

12.4.2 Registry

The best way of handling the registry may be to map it into a file-based representation on the server end. The Cfg and TCheetah plugins could then be used to set registry values as needed.

12.4.3 Files

For a first run there may be some way of utilizing [http://cygwin.com/ cygwin] to make use of the existing *nix POSIX module for manipulating files. There would probably need to be some changes to deal with the fact that open files can't be manipulated/moved/deleted at all in Windows (other than to do some registry magic that makes the changes on the next reboot).

12.4.4 Packages

Listing and removal of packages should be pretty easy via WMI. For installation in most cases the admin would need to figure out the correct silent install flags (there is a [http://www.appdeploy.com/ web site] that catalogs a lot of this information), and include that in the bcfg2 server-side XML along with a URL (like with the RPM plugin); the bcfg2 client itself would need to take care of download, perhaps via the [http://linux.duke.edu/projects/urlgrabber/ urlgrabber python module].

Another option would be to utilize one of the existing FLOSS tools for dealing with Windows packages, such as [http://wpkg.org/ WPKG].

12.4.5 Prior FLOSS Art

- [http://www.autoitscript.com/autoit3/ AutoIt] - For dealing with packages that don't have a silent install option
- [http://www.opensysadmin.com/trac/ticket/4 French Stuff]
- [http://ocsinventory.sourceforge.net/ Open Computers and Software Inventory - Next Generation]
- [http://www.glpi-project.org/spip.php?lang=en GLPI - Gestionnaire libre de parc informatique]
- Javascript thing a colleague of Desai's at ANL wrote - Desai was going to see if this can be released
- [http://sial.org/howto/cfengine/windows/ Managing Windows with CFEngine and Perl]
- [http://www.dmst.aueb.gr/dds/sw/outwit/ Outwit] - Small unixy utilities for Windows stuff like the registry and clipboard
- [http://www.cfengine.org/docs/cfengine-NT/ Porting cfengine to Windows NT]
- [http://isg.ee.ethz.ch/tools/realmen/ Real Men Don't Click] - Tobi Oetiker's stuff
- [http://isg.ee.ethz.ch/tools/realmen/res/index.en.html More Prior FLOSS Art]
- [http://unattended.sourceforge.net/ Unattended] - Bare Metal Installs, Package Management

- [http://wpkg.org/ WPKG] - Package Management

12.5 Writing Bcfg2 Specification

Bcfg2 specifications are logically divided in to three areas:

- Metadata
- Abstract
- Literal

The metadata portion of the configuration assigns a client to its profile group and to its non-profile groups. The profile group is assigned in Metadata/clients.xml and the non profile group assignments are in Metadata/groups.xml.

The group memberships contained in the metadata are then used to constuct an abstract configuration for the client. An abstract configuration for a client identifies the configuration entities (packages, configuration files, service, etc) that a client requires, but it does not identify them explicitly. For instance an abstract configuration may identify that a client needs the Bcfg2 package with

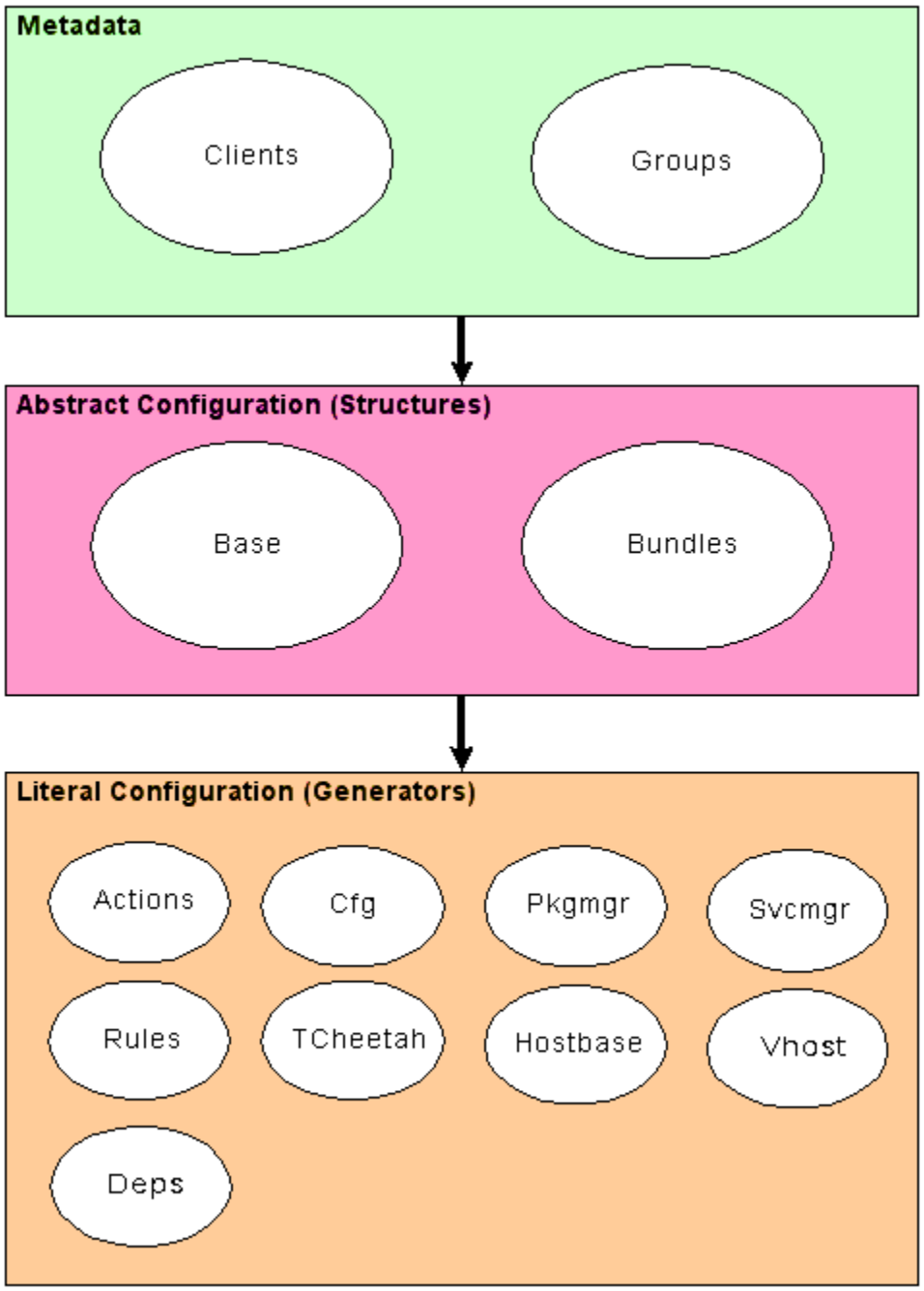
```
<Package name=bcfg2/>
```

but this does not explicitly identify that an RPM package version 0.9.2 should be loaded from http://rpm.repo.server/bcfg2-0.9.2-0.1.rpm. The abstract configuration is defined in the xml configuration files for the Base and Bundles plugins.

A combination of a clients metadata (group memberships) and abstract configuration is then used to generate the clients literal configuration. For instance the above abstract configuration entry may generate a literal configuration of

```
<Package name='bcfg2' version='0.9.2-0.1' type='yum'/>
```

A clients literal configuration is generated by a number of plugins that handle the different configuration entities.



12.5.1 Dynamic Groups

Dynamic groups are likewise complex, and are covered on their own [wiki:DynamicGroups page]

12.5.2 Abstract Configuration (Structures)

A clients Abstract Configuration is the inventory of configuration entities that should be installed on a client. Two plugins provide the basis for the abstract configuration, the Bundler and Base.

The plugin Bundler builds descriptions of interrelated configuration entities. These are typically used for the representation of services, or other complex groups of entities.

The Base provides a laundry list of configuration entities that need to be installed on hosts. These entities are independent from one another, and can be installed individually without worrying about the impact on other entities.

Usage of Groups in Base and Bundles

Groups are used by the Base and Bundles plugins for selecting Configuration Entity Types for inclusion in a clients abstract configuration. They can be thought of as:

```
if client is a member of group1 then
    assign to abstract config
```

Nested groups are conjunctive (logical and).:

```
if client is a member of group1 and group2 then
    assign to abstract config
```

Group membership maybe negated. See "Writing Bundles" for an example.

Configuration Entity Types

Entities in the abstract configuration (and correspondingly in the literal configuration) can have one of several types. In the abstract configuration, each of these entities only has a tag and the name attribute set.

The types of Configuration Entities that maybe assigned to the abstract configuration can be seen at *Configuration Entries*.

An example of each entity type is below.

```
<Package name='bcfg2'/>
<Path name='/etc/bcfg2.conf'/>
<Service name='ntpd'/>
<Action name='action_name'/>
```

Writing Bundles

Bundles consist of a set of configuration entities. These entities are grouped together due to a configuration-time interdependency. Basic services tend to be the simplest example of these. They normally consist of

- some software package(s)
- some configuration files
- an indication that some service should be activated

If any of these pieces are installed or updated, all should be rechecked and any associated services should be restarted.

All files in the Bundles/ subdirectory of the repository are processed. Each bundle must be defined in its own file and the filename must be the same as the bundle name with a .xml suffix.:

```
# ls Bundler
Glide3.xml
LPRng.xml
Tivoli-backup.xml
Tivoli.xml
a2ps.xml
abiword.xml
account.xml
adsm-client.xml
amihappy.xml
apache-basic.xml
apache.xml
apache2-basic.xml
apt-proxy.xml
at.xml
atftp-server.xml
atftp.xml
....
```

Groups can be used inside of bundles to differentiate which entries particular clients will receive. This is useful for the case where entries are named differently across systems; for example, one linux distro may have a package called openssh while another uses the name ssh. Configuration entries nested inside of Group elements only apply to clients who are a member of those groups; multiply nested groups must all apply.

Also, groups may be negated; entries included in such groups will only apply to clients who are not a member of said group.

When packages in a bundle are verified by the client toolset, the Paths included in the same bundle are taken into consideration. That is, a package will not fail verification from a Bcfg2 perspective if the package verification only failed because of configuration files that are defined in the same bundle.

The following is an annotated copy of a bundle:

```
<Bundle revision='$Revision: 2668 $' name='ssh' version='2.0'
        origin='https://svn.mcs.anl.gov/repos/bcfg/trunk/repository/Bundler/ssh.xml'>
  <Path name='/etc/ssh/ssh_host_dsa_key'/>
  <Path name='/etc/ssh/ssh_host_rsa_key'/>
  <Path name='/etc/ssh/ssh_host_dsa_key.pub'/>
  <Path name='/etc/ssh/ssh_host_rsa_key.pub'/>
  <Path name='/etc/ssh/ssh_host_key'/>
  <Path name='/etc/ssh/ssh_host_key.pub'/>
  <Path name='/etc/ssh/sshd_config'/>
  <Path name='/etc/ssh/ssh_config'/>
  <Path name='/etc/ssh/ssh_known_hosts'/>
  <Group name='rpm'>
    <Package name='openssh'/>
    <Package name='openssh-askpass'/>
    <Service name='sshd'/>
    <Group name='fedora' >
       <Group name='fc4' negate='true'>
```

```
        <Package name='openssh-clients'/>
      </Group>
      <Package name='openssh-server'/>
    </Group>
  </Group>
  <Group name='deb'>
    <Package name='ssh'/>
    <Service name='ssh'/>
  </Group>
</Bundle>
```

In this bundle, most of the entries are common to all systems. Clients in group "deb" get one extra package and service, while clients in group "rpm" get two extra packages and an extra service. In addition, clients in group "fedora" and group "rpm" get one extra package entries, unless they are not in the fc4 group, in which case, they get an extra package. Notice that this file doesn't describe which versions of these entries that clients should get, only that they should get them. (Admittedly, this example is slightly contrived, but demonstrates how group entries can be used in bundles)

|| '' ''Group '" '' || '' ''Entry '" '' || || all || /etc/ssh/ssh_host_dsa_key || || all || /etc/ssh/ssh_host_rsa_key || || all || /etc/ssh/ssh_host_dsa_key.pub || || all || /etc/ssh/ssh_host_rsa_key.pub || || all || /etc/ssh/ssh_host_key || || all || /etc/ssh/ssh_host_key.pub || || all || /etc/ssh/sshd_config || || all || /etc/ssh/ssh_config || || all || /etc/ssh/ssh_known_hosts || || rpm || Package openssh || || rpm || Package openssh-askpass || || rpm || Service sshd || || rpm and fedora || Package openssh-server || || rpm and fedora and not fc4 || Package openssh-clients || || deb || Package ssh || || deb || Service ssh ||

Bundle Tag

The Bundle Tag has the following possible attributes:

|| '' ''Name'' '' || '' ''Description'' '' || '' ''Values'' '' || || name || The name of the bundle || String || || version || Bundle schema version || String || || origin || URL of master version (for common repo) || String || || revision || Master version control revision || String ||

As mentioned above the Configuration Entity Tags may only have the name attribute in Bundle definitions.

Abstract Group Tag

In the Abstract Configuration plugins (Base and Bundle) the Group Tag may have the following attributes:

|| '' ''Name'' '' || '' ''Description'' '' || '' ''Values'' '' || || name || Name of group. || String || || negate || Negate the group association (is not a member of) || (True|False*) ||

An abstract group may contain any of the Configuration Entity types and other groups.

Using Base

The Base plugin provides a mechanism to add independent configuration entities to a client's abstract configuration. All files in the Base/ subdirectory of the repository are processed, and all entries that fall within

the scope of the client metadata are included in its abstract configuration.:

```
$ ls Base/
centos-4-x86.xml
fedora-core-4-x86.xml
rhel-as-4-x86.xml
rhel-es-4-x86.xml
rhel-ws-4-x86_64.xml
rhel-ws-4-x86.xml
```

```
<Base>
    <Group name='Centos4.4-Standard'>
        <Package name='audit'/>
        <Package name='audit-libs'/>
        <Package name='basesystem'/>
        <Package name='bash'/>
        <Package name='bcfg2'/>
        <Package name='beecrypt'/>
        ....
        <Package name='yum'/>
        <Package name='zlib'/>
        <Group name='x86_64'>
            <Package name='systemimager-x86_64initrd_template'/>
        </Group>
    </Group>
</Base>
```

The format of the Base files are similar to those used by the Bundler. The majority of the elements are usually Packages, but Paths of any type may all be defined. A partial example is below:

Base Tag

The Base Tag has no attributes

As mentioned above the Configuration Entity Tags contained in a Base definition may only have the name attribute in Base definitions.

Abstract Group Tag

In the Abstract Configuration plugins (Base and Bundle) the Group Tag may have the following attributes:

|| '' ''Name'' '' || '' ''Description'' '' || '' ''Values'' '' || || name || Name of group. || String || || negate || Negate the group association (is not a member of) || (True|False*) ||

An abstract group may contain any of the Configuration Entity types and other groups.

12.5.3 Literal Configuration (Generators)

A Generator is a Bcfg2 piece of code that is run to generate the literal configuration for a host using a combination of the hosts metadata and abstract configuration.

A Generator can take care of a particular configuration element. Any time this element is requested by the client, the server dynamically generates it either by crunching data and creating new information or by reading a file off of disk and passes it down to the client for installation.

Usage of Groups in Generators

Similar to Abstract Configuration plugins, groups are used by generator plugins for selecting Configuration Entities for inclusion in a clients literal configuration. They can be thought of as:

```
if client is a member of group1 then
    assign to abstract config
```

Nested groups are conjunctive (logical and).:

```
if client is a member of group1 and group2 then
    assign to abstract config
```

How the groups are configured is specific to the plugin, but here are two common methods:

- xml configuration file (Pkgmgr, Rules)
- file name encoding (Cfg, TCheetah, SSHBase)

Details are included on each plugin's page.

Generators

Each of the generators is covered on their own page.

|| '' ''Plugin'' '' || '' ''Description'' '' || || [wiki:Plugins/Actions Actions] || Action entries are commands that are executed either before bundle installation, after bundle installation or both. ||

CHAPTER

THIRTEEN

DEPRECATED/OBSOLETE DOCUMENTATION

These documents cover features that have been deprecated or that have been replaced in newer versions of Bcfg2. They're preserved here for folks using old versions of Bcfg2.

INDEX