Bcfg2 Manual

Narayan Desai

Rick Bradshaw

Joey Hagedorn

Bcfg2 Manual

by Narayan Desai

by Rick Bradshaw

by Joey Hagedorn

Published September 2005

Revision History

Revision 0.6.9 $Date: 05/09/02 10:20:52-05:00 $
$Id: manual.xml 1.9 05/09/02 10:20:52-05:00 desai@topaz.mcs.anl.gov $

Table of Contents

List of Examples

Chapter 1. Bcfg2 Architecture

Bcfg2 is based on a client-server architecture. The client is reponsible 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.

1.1. 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 sufficent 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 existance 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 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.

- Generators and administrative applications.
- Imcremental operations.

1.2. 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 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 of 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. Bundles groupings result in two different behaviors. Entries are assumed to be inter-dependant. 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

1.2.1. 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 specified in the configuration specification. The client merely includes a series of libraries while describe how to interact with the system tools on a particular platform.

Three of the libraries exist. There is the base set of functions, which contain definitions describing how to perform POSIX operations. Support for configuration files, directories, and symlinks 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.

1.3. 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. 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.

1.3.1. 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 (ie the tagname will be one of Package, ConfigurationFile, Service, Symlink, or Directory) 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 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 generators to provide these client-specific bindings.

The Bcfg2 server core contains a dispatch table that describes which generator can handle requests of a particular type. The responsible generator is located for each entry. It is called, passing in the configuration entry and the client's metadata. The behavior of generators is explicitly undefined, so as to allow maximum flexibility. The behavior of the stock generators is 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.

1.4. 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 notion of client-side operations. Without a static lexicon with defined semantics, this isn't possible.
- 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.

1.4.1. The Structure of Specifications

Configuration specifications contain some number of clauses. Two types of clauses exist. Bundles are groups of inter-dependant 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 contains 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. Five types of configuration entities exist: ConfigurationFile, Package, SymLink, Directory, and Service. 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. The easiest way to start using Bcfg2 is to write small static configurations for clients. Once configurations get larger, this process gets unwieldy; at this point, using the server makes more sense.

Chapter 2. Installing Bcfg2

2.1. Pre-requisites

Bcfg2 is written in python using several modules not included with most distributions. SSSlib, available from ftp://ftp.mcs.anl.gov/pub/sss/, provides communication abstraction. Element Tree, available from http://www.effbot.org provides convenient XML handling. Bcfg2 uses FAM (server-size) to coherently cache repository files and update them when they change.

ElementTree can be downloaded from http://www.effbot.org/downloads. It can be installed by running the setup script against the python installation.

```
$ python setup.py build
running build
running build_py
creating build
creating build/lib
creating build/lib/elementtree
copying elementtree/ElementInclude.py -> build/lib/elementtree
copying elementtree/ElementPath.py -> build/lib/elementtree
copying elementtree/ElementTree.py -> build/lib/elementtree
copying elementtree/HTMLTreeBuilder.py -> build/lib/elementtree
copying elementtree/SgmlopXMLTreeBuilder.py -> build/lib/elementtree
copying elementtree/SimpleXMLTreeBuilder.py -> build/lib/elementtree
copying elementtree/SimpleXMLWriter.py -> build/lib/elementtree
copying elementtree/TidyHTMLTreeBuilder.py -> build/lib/elementtree
copying elementtree/TidyTools.py -> build/lib/elementtree
copying elementtree/XMLTreeBuilder.py -> build/lib/elementtree
copying elementtree/__init__.py -> build/lib/elementtree
$ python setup.py install
...
```

SSSlib can be downloaded from ftp://ftp.mcs.anl.gov/pub/sss. It can either be built from source or prebuilt packages can be downloaded from the same location.

2.2. Bcfg2 Installation

2.3. Bcfg2 Initial Setup and Testing

Once the Bcfg2 software is installed, the configuration file and repository must be created. The example configuration file in `bcfg2/examples/bcfg2.conf` can be used, with minor modifications.

Example 2-1. bcfg2.conf

```
[server]
repository = /disks/bcfg2
structures = Bundler,Base
generators = SSHbase,Cfg,Pkgmgr,Svcmgr
metadata = /disks/bcfg2/etc
```

This configuration file sets the location of the configuration repository. It also activates two structures, and four generators. Structures are components that generate abstract configuration fragments. These are the form of the configuration. Generators provide client-specific values for each configuration settings contained in all abstract configuration fragments. Both of these are described in Section ???.

2.4. Daemon Configuration

Bcfg2 uses SSSlib, the communication libraries from the Scalable Systems Software project for communication abstraction. This library provides a unified messaging interface on top of several wire protocols with different authentication and encryption mechanisms. The default protocol is "challenge" which is a challenge response protocol with no data encryption. (SSL protection will be configured later). SSSlib also includes service location functionality; this allows software to locate components by name, regardless of their respective network locations. This function is provided with both static and dynamic implementations. Static component location setup will be sufficient for most Bcfg2 deployments.

Static component lookups depend on the file `/etc/sss.conf`. This file contains information about static service locations. This file must be the same on the server and all clients for communication to work properly. A location definition for the bcfg2 component will allow all clients to find and connect to it.

Example 2-2. /etc/sss.conf

```
<locations>
  <location component="bcfg2" host="bcfgserver"
  port="8052" protocol="challenge" schema_version="1.0" tier="1"/>
</locations>
```

This allows SSSlib to locate the bcfg2 component on the machine bcfgserver, port 8052, with the wire protocol "challenge".

2.5. New-Style XML-RPC Deployments

A new version of the Bcfg2 software is in testing that will provide simplified and standards compliant communications facilities. Instead of the use of SSSlib for communication, the server and clients can use HTTPS XML-RPC instead. This has required reimplementing the server and providing XML-RPC support for the client, but provides drastically simplified setup for new installs.

The prerequisite list now includes ElementTree, M2Crypto (for SSL functions) and Python 2.2 or newer. ElementTree and M2Crypto are both python modules that can be easily installed and are already packaged for many Linux distributions.

2.5.1. SSL Certificate Generation

SSL is used for channel-level data encryption. The requisite SSL certificates must be generated on the server side. I need to figure out how to do this.

2.5.2. Communication Bootstrapping

The Bcfg2 client must be able to find the server's location. This is accomplished through the use of the communication settings in `/etc/bcfg2.conf` Two settings for the this section are required: protocol and server url.

Example 2-3. Bcfg2 XML-RPC Communication Settings

```
[communication]
protocol = xmlrpc/ssl
url = https://localhost:9443
```

Chapter 3. Generators

Generators are modules which are loaded by the Bcfg2 server, based on directives in `/etc/bcfg2.conf`. They provide concrete, fully-specified configuration entries for clients. This chapter documents the function and usage of generators bundled with Bcfg2 releases. It also describes the interface used to communicate with generators; modeles implementing this interface can provide configuration elements for clients based on any representation or requirements that may exist.

3.1. Bundled Generators

This section describes the generators that come bundled with Bcfg2. As a general rule, generators requiring more than one configuration file will use a generator specific directory in the configuration repository.

3.1.1. Cfg

The Cfg generator provides a configuration file repository that uses literal file contents to provide client-tailored configuration file entries. The Cfg generator chooses which data to provide for a given client based on the aspect-based metadata system used for high-level client configuration.

The Cfg repository is structured much like the filesystem hierarchy being configured. Each configuration file being served has a corresponding directory in the configuration repository. These directories have the same relative path as the absolute path of the configuration file on the target system. For example, if Cfg was serving data for the configuration file `/etc/services`, then its directory would be in the relative path `./etc/services` inside of the Cfg repository.

Inside of this file-specific directory, three types of files may exist. Base files are complete instances of configuration file. Deltas are differences between a base file and the target file contents. Base files and deltas are tagged with metadata specifiers, which describe which groups of clients the fragment pertains to. Configuration files are constructed by finding the most specific base file and applying any more specific deltas.

Specifiers are embedded in fragment filenames. For example, in the fragment `services.C99_webserver`, "C99_webserver" is the specifier. This specifier applies to the class (C) webserver with a priority of 99. Other metadata categories which can be used include bundle (B), profile (P), hostname (H), attribute (A), and image (I). These are ordered from least to most specific: image, profile, class, bundle, and hostname. Global files are the least specific. Priorities are used as to break ties.

Info files, named `:info` are used to specify target configuration file metadata, such as owner, group and permissions. If no `:info` is provided, targets are installed with default information. Default metadata is root ownership, root group memberships, and 0644 file permissions.

Example 3-1. Cfg generator :info files

```
owner:root
group:root
perms:0755
```

Example 3-2. Cfg file repository example

```
$ ls
:info            passwd   passwd.C99_chiba-login
passwd.H_bio-debian  passwd.H_cvstest   passwd.H_foxtrot
passwd.H_reboot  passwd.H_rudy2    passwd.C99_netserv
passwd.B99_tacacs-server.cat  passwd.H_adenine
```

In the previous example, there exists files with each of the characteristics mentioned above. All files ending in ".cat" are deltas; ones with ".H_" are host specific files. There exists a base file, a `:info` file, two class-specified base files, and a bundle-specified base file.

3.1.2. The Scoped XML File Group: Servicemgr

The generator Servicemgr uses files formatted similarly to the metadata files. It works based on a single file, which contains definitions ordered by increasing specificity.

Example 3-3. `services.xml`

```
<Services>
<Class name='webserver'>
  <Service status='on' name='httpd' port='80' protocol='tcp'>
    <User address='0.0.0.0' mask='32'/>
  </Service>
</Class>
<Host name='mailhost'>
  <Service name='sendmail' status='on' protocol='tcp' port='80'>
    <User address='0.0.0.0' mask='32'/>
  </Service>
</Host>
<Host name='thai'>
  <Service name='ssh' status='off'/>
</Host>
<Service name='ssh' status='on' protocol='tcp' port='22'/>
<Service name='ntp-server' status='on' />
</Services>
```

This set of service definitions is intrepreted in the following way. Webservers run httpd, the host mailhost runs sendmail, and all machines run ssh, and the ntp-server.

3.2. The Generator API

The Bcfg2 core has a well-formed API used to call generators. This mechanism allows all stock generators to be runtime selected; no stock generators are required. The generator API has two main functions. The first is communication to the Bcfg2 core: the list of entries a particular generator can bind must be communicated to the core so that the proper generator can be called. The second function is the actual production of client-specific configuration element data; this data is then included in client configurations.

The inventory function is provided by a python dictionary, called __provides__ in each generator object. This dictionary has a key for each type of configuration entry (ConfigFile, Package, Directory, SymLink, Service), whose value is a dictionary indexed by configuration element name. For example, the data path to information about the service "sshd" could be reached at __provides__['Service']['sshd']. The value of each of these keys is a function that can be called to bind client-specific values to a configuration entry. This function is used in the next section.

The handler function located by the __provides__ dictionary is called with a static API. The function prototype for each of these handlers is:

Example 3-4. The Generator handler API

```
def Handler(self, entry, metadata):
    generator logic here
```

The data supplied upon handler invokation includes two parts. The first is the entry. This is a ElementTree.Element object, which already contains the configuration element type (ie Service) and name. All other data is bound into this object in this function. The range of data bound depends on the data type. The other data provided to handlers is client metadata, information about the current client, including hostname, image, profile, classes and bundles. The metadata is typically used to choose entry contents.

3.3. Writing a Generator

Writing a generator is a fairly straightforward task. At a high level, generators are instantiated by the Bcfg2 core, and then used to provide configuration entry contents. This means that the two points where control passes into a generator from Bcfg2 are during initial object instantiation, and every time a generator-provided configuration entry is bound.

Currently, generators must be written in python. They can perform arbitrary operations, hence, a generator could be written that executed logic in another language, but this functionality is currently not implemented.

Example 3-5. Simple Generator

```
from socket import gethostbyname, gaierror
from syslog import syslog, LOG_ERR
from Bcfg2.Server.Generator import Generator, DirectoryBacked, SingleXMLFileBacked, Generat

class Chiba(Generator):
    "'the Chiba generator builds the following files:
      -> /etc/network/interfaces"'

    __name__ = 'Chiba'
    __version__ = '$Id: generators.xml 1.5 05/06/01 12:17:25-05:00 desai@topaz.mcs.anl.gov
    __author__ = 'bcfg-dev@mcs.anl.gov'
    __provides__ = {'ConfigFile':{}}

    def __init__(self, core, datastore):
        Generator.__init__(self, core, datastore)
        self.repo = DirectoryBacked(self.data, self.core.fam)
        self.__provides__['ConfigFile']['/etc/network/interfaces'] = self.build_interfaces

    def build_interfaces(self, entry, metadata):
        "'build network configs for clients"'
        entry.attrib['owner'] = 'root'
        entry.attrib['group'] = 'root'
        entry.attrib['perms'] = '0644'
        try:
            myriaddr = gethostbyname("%s-myr" % metadata.hostname)
        except gaierror:
            syslog(LOG_ERR, "Failed to resolve %s-myr"% metadata.hostname)
            raise GeneratorError, ("%s-myr" % metadata.hostname, 'lookup')
        entry.text = self.repo.entries['interfaces-template'].data % myriaddr
```

Generators must subclass the Bcfg2.Server.Generator.Generator class. Generator constructors must take two arguments: an instance of a Bcfg2.Core object, and a location for a datastore. __name__, __version__, __author__, and __provides__ are used to describe what the generator is and how it works. __provides__ describes a set of configuration entries that can be provided by the generator, and a set of handlers that can bind in the proper data. build_interfaces is an example of a handler. It gets client metadata and an configuration entry passed in, and binds data into entry as appropriate.

Chapter 4. Deploying Bcfg2

Bcfg2 can be deployed in several different ways. The strategy chosen varies based on the level of complexity accepted by the administrators. The more literal a representation used, the less powerful and reusable it is. We will describe three strategies for Bcfg2 deployment, ranging from a cfengine-like deployment, to a highly abstract configuration. While the abstract configuration is much more powerful, the cfengine-like deployment is much easier to understand and manipulate.

4.1. Simple Deployments

The Bcfg2 server will build configurations based on a set of high-level specifications that use class-based abstractions to provide reusability. This approach works pretty well; however, it can be hard to deploy and may be too complicated to solve simple problems.

This issue can be addressed through the use of the Bcfg2 client with a static configuration specification. This method works as follows: important configuration details are statically specified in a file on each system. The Bcfg2 client runs periodically, and ensures that all aspects of configuration included in the static specification are correct. It then performs any update operations needed on the client.

The format of static specifications is identical to that provided by the Bcfg2 server, when it is used. It consists of a series of "Bundle" and "Independent" clauses. Independent clauses contain a series of configuration elements that can be installed without any install time dependence on other configuration elements. Bundles are series of dependent clauses. This means that configuration elements may interfere with one another, or that services may need to be restarted upon configuration update.

Each of these containers consists of a series of configuration elements. The same elements may appear in either type of clauses. These are basic types that are the same across all OS ports.

Package

A software package. This entity includes a package name and version number. It may optionally include installation information (such as a package source URL) if one is needed.

ConfigFile

A configuration file. This entity includes a file path, owner, group, permissions, and file contents.

SymLink

A symbolic link. This entity includes a source and destination.

Service

A service (de)activation. This controls services, a la chkconfig or update-rc.d. Services are restarted whenever co-located configuration entities are modified. This ensures that any configuration changes are flushed out to all active processes.

Directory

A filesystem directory. This entity includes an owner, group and permissions.

4.2. A Near-Literal Deployment

The next easiest method to deploy is one where the configuration specification is as simple and literal as possible. This style of configuration specification can be characterized as near copies of parts of the system.

This style of deployment uses the stock generators: Cfg, Pkgmgr, and Svcmgr. These manage configuration files, packages and services, respectively. Copies of configuration files are placed in the Cfg repository, in as generic a location as possible.

4.3. An Abstract Deployment

4.4. Object Oriented Configs

One of the most powerful and useful parts about bcfg2 metadata system is the ability to have truely Object Oriented configs. I have found that this has made me understand the machines I am managing in a whole new light. Instead of focusing on what is installed, I now focus on how machines relate to each other, or what pieces of the metadata are similar in their configs. To illistrate this think about the following example machines:

- A users desktop machine
- A multiuser compute machine
- A users home machine. (telecommuter)

These 3 machines have 3 distinct focuses for usage, but in reality they can have a very similar metadata, depending on how the config is broken up or the view that is taken of the config. If you focus first on where the machines are the same it will help to build the common metadata. below is what all 3 machines have in common:

- users need to have the software they need to perform there work.

let just encode this into metadata by creating a Class called common-software that contains all the common software between all 3 machines.

```
<Class name='common-software'>
```

```
  <Bundle ... />
  ....
</Class>
```

now we need to find where they differ:

Desktop machines

- need to have a GUI interface(Xwindows or what not)
- use NIS for authentification
- use Autofs to mount home directories

Multiuser compute machines

- only accessible by SSH, No GUI interface
- use NIS for authentification
- use Autofs to mount home directories

Home machine

- need to have a GUI interface(Xwindows or what not)
- use static password file
- use local disk for home directories

As you can see that there are common things pairwise in these configs that can be further exploited by the object oriented system of bcfg2

```
  <Class name="Gui-Interface">
  <Bundle ... />
  ...
</Class>
  <Class name="NIS-Autofs">
  <Bundle ... />
  ...
</Class>
```

now all that is left is to ensure that all needs are met and we find we need to make one more class:

```
  <Class name="static-password-local-disk">
  <Bundle ... />
  ...
</Class>
```

Now we can mix and match these classes together to build the 3 profiles or even build new profiles with these descrete entities.

```
  <Profile name='desktop'>
  <Class name='common-software'/>
  <Class name="Gui-Interface"/>
  <Class name="NIS-Autofs">
</Profile>
  <Profile name='computerserver'>
  <Class name='common-software'/>
  <Class name="NIS-Autofs">
</Profile>
  <Profile name='home-desktop'>
  <Class name='common-software'/>
  <Class name="Gui-Interface"/>
  <Class name="static-password-local-disk">
</Profile>
  <Profile name='bare-system'>
  <Class name='common-software'/>
</Profile>
```

The free form object oriented fashion in which metadata can be constructed is truely a double edge sword. On one hand you can build up a nice list of descrete entities that can compose even the most complicated configs, but it also allows for the creation of entities that could provide monolithic solutions to each machines config. It is all in how one views the machines and how much they are willing to harness the power of the OO based metadata system.

4.5. Tips & Tricks

4.6. An example application of bcfg2

In my computing environment there are quite a diverse set of machines and requirements for there operation. What this meant is that I needed to devise a build system for machines that would allow me to easily customize the software and services on the machine while still being able to easily manage them and keep them secure. What I came up with that solved this problem was that the initial install needed to be the smallest subset of software that all machines had in common and install this with whatever automated install system fit the OS. The goal being that the OS automated installer(ie: kickstart, or systemimager) would put the initial bits on disk and take care of hardware stuff and then as part of the postinstall process I run bcfg2 to insure that the rest of the software and configuration occurs based on the machines metadata. The overall goal was met. I could now build any type of machine that I needed just by using the common buildsystem and let bcfg2 determine what was different machine to machine.

My current build process is centered around systemimager and bcfg2. I have done some small enhancements to systemimager so that with one floppy or cdrom any administrator can build any number of machine profiles automatically. This is all done with some of the new features that allow the encoding of the profile and image in the clientside command so that the back end metadata can be asserted from the client, which overrides the defaults specified in the metadata.xml file.

Chapter 5. The BCFG2 Reporting System

5.1. Reporting Quick Start

Example 5-1. `report-configuration.xml`

```
<Reports>
      <Report name='core_stats' good='Y' modified='Y'>
          <Delivery mechanism='mail' type='nodes-digest'>
              <Destination address='user@domain.tld'/>
              <Destination address='user@otherdomain.tld'/>
          </Delivery>
          <Delivery mechanism='www' type='nodes-digest'>
              <Destination address=''/var/www/core_stats.html'/>
          </Delivery>
          <Machine name='.*'/>
      </Report>

      <Report name='stats_for_a_machines' good='N' modified='Y'>
          <Delivery mechanism='mail' type='nodes-digest'>
              <Destination address='user@domain.tld'/>
          </Delivery>
          <Delivery mechanism='mail' type='overview-stats'>
              <Destination address='user@otherdomain.tld'/>
          </Delivery>
          <Machine name='a.*'/>
          <Machine name='x-aim'/>
      </Report>
    </Reports>
```

This configuration will generate two separate reports and deliver them a number of different ways. For more information on exactly what each section does, please refer to the Configuration section below

In order to run reports, you need to run the following command regularly via **Cron**: **GenerateHostInfo**

Once configured correctly, just wait for the e-mail or check look at the output html files with a web browser. You can run **GenerateHostInfo** by hand if you would like in order to try it out immediately. If you are trying to view a HTML file and find that it doesn't work as expected, make sure that the required source directory is in the same directory. They are static, but might need to be moved from prefix/share/bcfg2/ in to the directory from which you are viewing your reports.

5.2. Concepts; Why to report

Reports play an important role in effectively managing systems with BCFG. There are two primary functions they fulfill; providing otherwise unobtainable information, and presenting additional helpful information to allow for easier admistration. Reports can contain information including system statistics, discrepancies between specified and actual configuration, invalid configuration, and auditing information among other things.

The flexible XML configuration file allows reports to be configured to deliver only the information that is important. Additional reports can easily be created, providing site-specific capability to manage at record effiency. The capability to harvest information regarding statistics, configuration, and problems in a single location should prove to be powerful.

5.3. How it all works

The BCFG2 Reporting System consists of a number of elements. The core is the **StatReports** Executable. **StatReports** reads a default configuration file (or a config file specified on the command line) then prepares and delivers the reports specified in the configuration file. It is expected that this executable will be run by the adminstrator periodically via **cron** or similar facilty. The executable can also be run manually on demand for a special sort of report that needs to be generated immediately.

StatReports gets the data it reports from a number of sources. The `hostinfo.xml` file is specific to the reporting system and generated by the executable python script **GenerateHostInfo** . `Hostinfo.xml` contains information about if a host is currently pingable or not, and a mapping of short hostnames to FQDNs. **GenerateHostInfo** will be run automatically by **StatReports** if the `hostinfo.xml` file is older than 23.5 hours. This number is chosen, because it is likely an administrator will recieve reports daily about the status of hosts and if they had run BCFG the previous night. It is possible to execute **GenerateHostInfo** to update the `hostinfo.xml` file at any interval via cron, but it does take some time to complete, so be sure to give it a few minutes. This will only likely be of use if your BCFG clients are set to update more often than nightly and you would like reports after each run.

The next place **StatReports** gets data from is the `statistics.xml` file. This file is maintained by bcfgd, and is updated whenever a client updates, therefore is always up to date, and no maintainence is required on this file.

Finally **StatReports** is able to get any information out of the `metadata.xml` file. This will likely be most useful in the future where the range of reports will include auditing style reports, that must include configuration information.

5.4. Report Types

There are a number of report types, and a number of delivery styles. It is expected that reports be laid out around a group of machines. For any group of machines it can be defined that there be any number of reports generated, with different options. For each of those reports, each can be delivered by Mail, WWW, or via RSS (or any combination of the three.) In the future, additional report types will be added, and if necessary, additional types of deliveries will be created.

Here is a list of the report types currently defined.

Overview-Stats

This report provides informatoin about a large number of machines and their states. It is often found to be useful when the group of machines it is connected with is simply All Nodes, which gives an overall outlook on your network's health. It makes sense to get this report via any mechanism

Nodes-Digest

This is a report that includes details about each node, specifically what packages, files, etc are broken, and other node specific info. It makes sense to recieve this via any mechanism

Nodes-Individual

This report includes details about each node, but information is separated in to separate sections (such as separate e-mails or RSS articles) before sending. This works well with e-mail filters and error detection. Currently WWW is not a supported delivery mechanism for this type of report, because it is not completely clear how such a report could be used.

A list of delivery mechanisms follows:

www

This delivery produces HTML reports which can be delivered via the WWW. it is important that you copy the directory: <prefix>/share/bcfg2/reports/web-rprt-srcs in to the same directory as you will be serving web pages from. It includes CSS and JavaScript files necessary to properly view the WWW files.

rss

This delivery type also should be written to some area that is likely web accessable. It is plain RSS-- One Caveat-- the "Article Link" does not actually point to any web information, because it is not clear that a corresponding web page exists for any given RSS article.

mail

Mail is simple plaintext e-mail that is sent to the specified recipients directly from the machine on which BCFG is running by opening a pipe to sendmail.

5.5. Configuration

The `report-configuration.xml` file is the standard file that is used whenever the **StatReports** executable is run without any command line arguments. Alternate configuration files, formatted identically, can be used instead with the -c flag. This can be useful for running different types of reports at different intervals. For example:

```
Run this hourly: StatReports -c WebAndRssReport-config.xml
Run this daily: StatReports -c emailReports-config.xml
```

The `report-configuration.xml` file is structured with a root <Reports/> tag at the top level. Within this tag any number of <Report/> tags can be inserted. Each report is structured around a group of machines. <Machine/> tags may individually reference a machine by hostname (not FQDN), or also by a Python Regular Expression. More information can be found about such Regexes at: http://docs.python.org/lib/re-syntax.html.

Any number of <Delivery/> elements can be made for a given report. A delivery consists of a mechanism and a type. The mechanism would be something like Mail or Web, and they type would reference the content of the report. Some are tailored to overall machine health, while others could be best fit for auditing purposes

Finally, each <Delivery/> contains one or more <Destination/> tags. In the case of an RSS or WWW report, the destination should be a complete path to the output file. In e-mail based reports the destination should be a complete e-mail address.