Bcfg2 Manual

Narayan Desai, Argonne National Laboratory <desai@mcs.anl.gov>
Rick Bradshaw, Argonne National Laboratory
<bradshaw@mcs.anl.gov>
Joey Hagedorn, Argonne National Laboratory
<hagedorn@mcs.anl.gov>

Bcfg2 Manual

by Narayan Desai, Rick Bradshaw, and Joey Hagedorn

$Revision: 2261 $

Published July 2006

Table of Contents

1. Bcfg2 Architecture 1
    - Goals 1
    - The Bcfg2 Client 1
        - Architecture Abstraction 3
    - The Bcfg2 Server 3
        - The Configuration Specification Construction Process 3
    - The Literal Configuration Specification 4
        - The Structure of Specifications 4
    - Design Considerations 5
        - System Metadata 5
        - Package Management 6
2. Installing Bcfg2 7
    - Pre-requisites 7
    - Bcfg2 Initial Setup and Testing 7
    - Daemon Configuration 8
        - SSL Certificate Generation 8
        - Client Communication Setup 8
3. Writing Bcfg2 Specifications 9
    - Interacting with Client Groups in Bcfg2 9
    - Adding to the Abstract Configuration 10
        - Writing Bundles 10
        - Using Base 10
    - Adding to the Literal Configuration 11
        - Cfg 11
        - Pkgmgr 12
        - Svcmgr 12
        - Rules 13
        - SSHbase 13
    - Checking Group-External Clients for Unintended Changes 13
    - Validating the Bcfg2 Repository 13
    - Annotated Configuration Examples 13
        - Configuring /etc/motd on all hosts 13
        - Configuring NTP for a network 14
4. Deploying Bcfg2 15
    - Simple Deployments 15
    - A Near-Literal Deployment 15
    - An Abstract Deployment 15
    - Bcfg2 Server Administration 15
    - An example application of bcfg2 15
5. Developing for Bcfg2 17
    - Bcfg2 Plugins 17
    - Writing Bcfg2 Plugins 17
        - An Example Plugin 18
6. BCFG2 Reports 20
    - How it works 20
    - Report Types 20
    - Configuration 21
    - Reporting Quick Start 21

List of Tables

2.1. Bcfg2 Software Prerequisites 7
3.1. Bcfg2 Group Parameters 9
3.2. Bcfg2 Configuration Entity Types 10
3.3. Package Entity Attributes 12
5.1. Bcfg2 Plugin Functions 17
5.2. Bcfg2 Plugin Members 17
6.1. Bcfg2 Report Types 20
6.2. Bcfg2 Report Delivery Mechanisms 21

List of Examples

2.1. /etc/bcfg2.conf 7
2.2. /etc/bcfg2.conf 8
3.1. Cfg/filepath/:info 11
3.2. Cfg/etc/passwd/ 11
3.3. Svcmgr/ssh.xml 12
5.1. A Simple Plugin 18
6.1. etc/report-configuration.xml 22

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.

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.
- Plugins and administrative applications.
- Imcremental operations.

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:

1. 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.

2. 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 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.

3. 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-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.

4. 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

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 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, 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.

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.

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:

1. 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.

2. 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.

3. 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 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 behaviors 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.

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.

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.

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.

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.

1. 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.

2. Creating a specialized version of a currently 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.

3. 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:

1. 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.

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 2. Installing Bcfg2

Pre-requisites

Bcfg2 is written in python using several modules not included with most distributions. lxml provides convenient xml handling.

The Bcfg2 server requires a few more packages. It uses either FAM or Gamin to coherently cache repository files and update them when they change. It also requires pyOpenSSL to use SSL functions.

lxml is required for xml parsing. It can be downloaded from http://www.codespeak.net/lxml. It, in turn, requires libxml2, libxslt, and pyrex.

The python fam binding can be downloaded from python-fam.sourceforge.net. FAM (on several linux distributions) has been depricated in favor of gamin. The Bcfg server will autodetect which modules are available, and use appropriate file caching logic. It can be installed by running the setup.py script.

Table 2.1. Bcfg2 Software Prerequisites

Name	Description	URL
lxml	XML Processing	http://codespeak.net/lxml
pyrex	C to Python language interoperability (needed for lxml)	http://www.cosc.canterbury.ac.nz/~greg/python/Pyrex
pyOpenSSL	OpenSSL bindings for Python	http://pyopenssl.sourceforge.net/
Fam	File Alteration Monitor	http://oss.sgi.com
Gamin	Alternate File Alteration Monitor	http://www.gnome.org/~veillard/gamin/
Python-fam	Python bindings for fam (not needed with gamin)	http://python-fam.sourceforge.net

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. This should be placed in `/etc/bcfg2.conf`. If it is placed in another location, each program takes a command line argument to specify its alternate location.

Example 2.1. /etc/bcfg2.conf

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

This configuration file sets the top level location of the configuration repository. It also activates two structures, and four generators. Both structures and generators are instances of Bcfg2 server plugins. Structures generate abstract configuration fragments. These form the inventory of the configuration. Generators provide client-specific literal values for each configuration entity contained in the abstract configuration.

Daemon Configuration

Bcfg2 uses XML-RPC over HTTPS for all communications. All communications occur over this transport. HTTPS provides data security, while an embedded username and password provide authentication.

SSL Certificate Generation

SSL is used for channel-level data encryption. The requisite SSL certificates must be generated on the server side. The following command will generate a server key:

```
openssl req -x509 -nodes -days 1000 -newkey rsa:1024 \
        -out bcfg2.key -keyout bcfg2.key
```

This command will generate an SSL key including both an RSA key and a certificate. This is suitable for use with the Bcfg2 server. The path to this key should be put in the bcfg2 configuration file in section communication, setting key.

Client Communication Setup

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` Several settings must be included in this file: the server url, a username and a password.

Example 2.2. /etc/bcfg2.conf

```
[communication]
protocol = xmlrpc/ssl
password = pwd
user = root

[components]
bcfg2 = https://bcfg2server:8765
```

Chapter 3. Writing Bcfg2 Specifications

The Bcfg2 specification is a set of directives that describe how hosts should be configured. This information is used to generate client configurations. This section describes the steps taken during the composition of bcfg2 specifications.

1. All parts of the specification will correspond to some subset of the clients that bcfg2 has record of. Find or create a group corresponding to the target of this specification.
2. Add each new configuration entry to the "abstract configuration" for the target group. This can be done in one of two ways. If the new configuration has to do with a service, or some other piece of inter-dependent configuration, write a bundle. If not, add the extra configuration entries to the base.
3. Add configuration data for the above entries that apply only to the target group.
4. Verify that clients not in the target group remain unaffected by these changes.
5. Re-validate the bcfg2 repository by running **bcfg2-repo-validate**.

Interacting with Client Groups in Bcfg2

Bcfg2 uses an aspect-based classing mechanism to describe configuration patterns in its specifications. Each class describes particular aspects of client configurations. The Bcfg2 metadata mechanism has two types of information, client metadata and group metadata. Client metadata describes what top level group a client is associated with. Its configuration is derived from a combination of its host information and this group. Group definitions describe groups in terms of what bundles they include and other groups they include. Groups have a set of properties that describe how they can be used. (Starred values are defaults)

Table 3.1. Bcfg2 Group Parameters

Name	Description	Values
profile	If a client can be directly associated with this group	(True|False*)
public	If a client can freely associate itself with this group	(True|False*)
toolset	Describes which client-side logic should be used to make configuration changes	(rh|debian|solaris|aix|auto)
category	A group can only contain one instance of a group in any 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

When a client's configuration is generated, its metadata is fetched. This includes a list of all groups recursively dereferenced, and all bundles included by those groups. This collection has already been processed using the group category rules, so only one instance from each group category is included. This metadata is used throughout the rest of the configuration generation process; it defines the client's abstract configuration and specifies all literal contents of all configuration entities.

Adding to the Abstract Configuration

When writing bcfg2 specification, administrators primarily perform one of two operations: addition of new configuration entities or the modification of existing entries. If new entities need to be added, then they must be added to the abstract configuration. This 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 bundler builds descriptions of interrelated configuration entities. These are typically used for the representation of services, or other complex groups of entities. 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.

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.

Table 3.2. Bcfg2 Configuration Entity Types

Name	Description
Package	Software Package
ConfigFile	Configuration File
Service	Persistent system services and daemons
Directory	Filesystem Directories
SymLink	Symbolic links
Permissions	The permissions (not contents) of a POSIX path

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 consist of some software package(s) some configuration files and 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.

Bundles can also contain conditonal entries that are only used for hosts in some particular groups. This is useful when a service name varies from platform to platform. A group is defined for each platform, hence a different service can be associated with the bundle for clients in the different groups. Conditional additions can also add extra functionality to services as needed. This has proven useful in the case of configuring webservers; php and ssl can be configured as extra features that some webservers have and others do not. At the same time, all configuration-time interdependencies are maintained.

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 respository are processed, and all entries that fall within the scope of the client metadata are included in its abstract configuration. These files are similar to those used by the Bundler, Svcmgr, and Pkgmgr, without the need for prioritization used by the later two.

Adding to the Literal Configuration

During the construction of the literal configuration, first the abstract configuration is built, and then explicit data is bound in to each entity. The previous section describes how the first stage works, and this section describes the second stage. Each entity will be served by one plugin. This plugin will decide what explicit data should be bound in to a particular entity for a given client. Each of these plugins has a specific area of the configuration repository, corresponding to its name. This section describes how several of the basic plugins works.

Cfg

The Cfg plugin provides a configuration file repository that uses literal file contents to provide client-tailored configuration file entries. It 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.G99_webserver`, "G99_webserver" is the specifier. This specifier applies to the group (G) webserver with a priority of 99. Files can also be tagged with a host-specific (H) specifier.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. This file can also contain an encoding parameter (ascii|base64) and a paranoid flag that causes diffs to be logged on clients.

Example 3.1. Cfg/filepath/:info

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

Example 3.2. Cfg/etc/passwd/

```
$ ls
passwd.H_adenine     passwd            passwd.G99_chiba
passwd.H_bio-debian  passwd.H_cvstest  passwd.H_foxtrot
passwd.H_reboot      passwd.H_rudy2    passwd.G98_netserv
passwd.G99_tacacs-server.cat           :info
```

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.

Pkgmgr

The Pkgmgr plugin is responsible for providing package version and installation information. In the case of each "Package" entity in the configuration, it binds in information needed to detect, verify and install the package. It has a similar format to the files used by Base and Bundler, but with a few differences. First, each file has a priority. This allows the same entity 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. The other difference is that automatic deriviation of package information from the file attribute. The Pkgmgr has a set of regular expressions that can split package names for several formats. The filenames are used to construct installation URLs, and set several important fields like package name and version.

Table 3.3. Package Entity Attributes

Name	Description
name	Package Name
version	Package Version
uri	URL-style location of file repository (typically http)
file	Package file name. Several other attributes (name, version, url) can be automatically defined based on regular expressions definied in the Pkgmgr plugin.
simplefile	Package file name. No name parsing is performed, so no extra fields get set

Svcmgr

The Svcmgr plugin describes where services should be active and inactive. Its files have a similar form to those used by the Pkgmgr. Several files in the Svcmgr repository can contain overlapping definitions, and a per-file priority is used to determine precedence, the highest priority file serving data for a particular service wins, on a service by service basis.

These files also have a similar set of semantics to those used by the Pkgmgr. Entries in the top level element (Services) are global definitions. Group elements describe additional conditions that must be matched for that definition to supercede less specific ones. Deeply nested definitions must have their parent condition matched, plus all parent conditions as well. For example, the following declaration turns ssh on by default, disables it if the client is a part of group a, and reenables it if the client is a part of both groups a and b. Group nesting provides a conjunctive function.

Example 3.3. Svcmgr/ssh.xml

```
<Services priority='0'>
  <Service name='ssh' status='on'/>
  <Group name='a'>
    <Service name='ssh' status='off'/>
    <Group name='b'>
```

```
      <Service name='ssh' status='on'/>
    </Group>
  </Group>
</Services>
```

The files used by this plugin can be structured in a number of ways. The most common method is to use one large file, but this can be inconvenience due to the large size of the file. The data can also be split up using any convenient mechanism: per-service, per-administrator, etc.

Rules

The Rules plugin works like Pkgmgr and Svcmgr, but can be used for any entries, including literal packages and services, directories, permissions and symlinks.

SSHbase

The SSHbase plugin implements ssh public and private key management functionality. This means that a central record of ssh host keys is maintained. Also, a correct ssh_known_hosts file is maintained. This means that the keys for new hosts are added to this configuration, and also that a correct line for localhost is created. SSHbase will generate a new key for any hosts that doesn't already have a key stored in the repository, so it should be pre-seeded with the keys (public and private) of pre-existing clients.

Checking Group-External Clients for Unintended Changes

Any configuration change will apply to some set of clients. Often, repository changes can have unintended consequences to clients not included in the target group. To address this issue, consider the changes performed, and if they can affect clients in unexpected ways.

Validating the Bcfg2 Repository

Bcfg2 includes a repository validation tool that will check all XML files in the repository against included XML schemas. It is critical to run this command, **bcfg2-repo-valdate** after any modifications to XML files. If all files validate properly, then no output will be returned. It takes a "-v" option that prints out a line for each file that is validated. This can be used to ensure that all files are checked.

Annotated Configuration Examples

In addition to the description of the abstract process above, we present several examples of the thought process and actions taken to achieve a particular configuration goal. These will start simple, but become more complex.

Configuring /etc/motd on all hosts

The goal for this example is to install a uniform copy of a specified `/etc/motd` on all hosts.

1. In this case, the target group is all clients, since we want this version of `/etc/motd`. As

mentioned earlier, the global group is handled specially, so that all new clients, even newly created ones, are in it.

2. Since `/etc/motd` is not interdependent with any other configuration entities, it can be installed using Base instead of using Bundler. The ConfigFile entity should be placed in the globally scoped section of a base file. This adds the configuration file to the abstract configuration.

3. A file with the correct contents for `/etc/motd` should be installed in the Cfg repository area as a global file. This will provide the right literal configuration specification for each client.

4. Since this change is globally scoped, there are not any clients that should not be affected.

5. Finally, **bcfg2-repo-validate** should be run to catch typos.

Configuring NTP for a network

The goal for this example is to configure NTP for an entire network. This implies several things. All clients should run NTP as clients. Some hosts should run NTP as a server, and other hosts should use local NTP service instead of an external server.

1. Two discrete groups are used for the different parts of the desired configuration state. The first is the global group, handling the "all clients run ntp" part of the configuration specification. The other part corresponds to a new group "ntp-server", which contains only clients that should function as an ntp server.

2. This configuration specification describes configuration entities that have interdependencies, so a bundle should be used. This bundle should contain the ntp software, the ntp configuration file, and the ntpd service.

3. The literal portions of three different configuration entities need to be represented. First, the Pkgmgr needs to be configured to bind a package entity named ntp. Next, the Svcmgr needs to have a global declaration that the service ntpd should be on. Finally, two different configuration files should be added to Cfg. The global version of `/etc/ntp.conf` should have an ntp configuration that points hosts at the local ntp server. The group "ntp-server" specific version of this file should have the proper configuration for local ntp servers.

4. A quick inspection of these configuration changes show only minor possibilities for bad interactions. All machines should run ntp, and the only scope where bad interactions can occur is on ntp servers.

5. Finally, run **bcfg2-repo-validate**. It will validate the new bundle that has been added, and changes to the Svcmgr and Pkgmgr indices.

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.

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.

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.

An Abstract Deployment

Bcfg2 Server Administration

An example application of bcfg2

In my computing environment there are quite a diverse set of machines and requirements for

their operation. What this meant was 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. Developing for Bcfg2

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.

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.

Table 5.1. Bcfg2 Plugin Functions

Name	Description
Probes	Plugins can send executable code to clients, where local contributions to configuration state can be gathered.
Abstract Configuration Structures	A plugin can define new groups of interdependent and independent configuration entities
Literal Configuration Entities	Plugins can provide literal configuration entity information.
XML-RPC Functions	Plugins can expose a set of functions through the Bcfg2 server's authenticated XML-RPC interface.

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.

Table 5.2. Bcfg2 Plugin Members

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
__rmi__	Set of functions to be exposed as XML-RPC functions	List of function names (strings)
Entries	Multidimentional dictionary of keys that point	Dictionary of Configuration-

Name	Description	Format
	to the function used to bind literal contents for a given configuration entity.	EntityType, 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

An Example Plugin

Example 5.1. A Simple Plugin

```
import socket, Bcfg2.Server.Plugin

class Chiba(Bcfg2.Server.Plugin.Plugin):
    '''the Chiba plugin builds the following files:
      -> /etc/network/interfaces'''

    __name__ = 'Chiba'
    __version__ = '$Id: chiba.py 1702 2006-01-19 20:20:51Z desai '
    __author__ = 'bcfg-dev@mcs.anl.gov'
    Entries = {'ConfigFile':{}}

    def __init__(self, core, datastore):
        Bcfg2.Server.Plugin.Plugin.__init__(self, core, datastore)
        self.repo = Bcfg2.Server.Plugin.DirectoryBacked(self.data,
          self.core.fam)
        self.Entries['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 = socket.gethostbyname("%s-myr" % \
          metadata.hostname)
        except socket.gaierror:
            self.LogError("Failed to resolve %s-myr"% metadata.hostname)
            raise Bcfg2.Server.Plugin.PluginExecutionError, ("%s-myr" \
            % metadata.hostname, 'lookup')
        entry.text = self.repo.entries['interfaces-template'].data % \
          myriaddr
```

Bcfg2 server plugins must subclass the Bcfg2.Server.Plugin.Plugin class. Plugin constructors must take two arguments: an instance of a Bcfg2.Core object, and a location for a datastore. __name__, __version__, __author__, and Entries are used to describe what the plugin is and how it works. Entries 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. This results in a `/etc/network/interfaces` file that has static information derived from DNS for a given host.

Chapter 6. BCFG2 Reports

Reports play an important role in effectively managing systems with BCFG. There are two primary functions they fulfill; providing otherwise unobtainable information, and presenting common information in a compact, effective format that allows for easier admistration. Reports can contain system statistics, discrepancies between specified and actual configuration, invalid configuration notices, 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.

How it works

The BCFG2 Reporting System consists of a number of elements including the **bcfg2-build-reports** Executable, a configuration file, and XSLT transform files. **bcfg2-build-reports** reads a default configuration file (or a config file specified on the command line) then prepares and delivers the reports according to the format defined in the transform files. It is expected that this executable will be run by the adminstrator periodically via **cron** or similar facility. The executable can also be run manually on demand for a special sort of report that needs to be generated immediately.

bcfg2-build-reports gets the data it reports from a number of sources. `Metadata/clients.xml` contains information about if a host is currently pingable or not. **bcfg2-ping-sweep** will be run automatically by **bcfg2-build-reports** if the `Metadata/clients.xml` file is out of date with pingability information.

The next place **bcfg2-build-reports** 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 maintainance is required on this file. Most of the information in the predefined reports come from this file.

Finally **bcfg2-build-reports** is able to pull information from the `Metadata/groups.xml` file as well. This allows reports to describe the configured profile for each client.

Report Types

There are a number of report types and delivery styles to present and transmit the reported data. The reporting structure lends itself best to structuring reports around groups of machines. For any group of machines any number of reports are generated. Each report may be delivered via Mail, WWW, or 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. It is easy to create your own custom report using XSLT. Tables describing report types and report delivery mechanisms follow:

Table 6.1. Bcfg2 Report Types

Report Type	Description
Overview-Stats	This report provides information about a large number of machines and their states. It is often found to be useful when the constituent machines

Report Type	Description
	are simply specified as 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 report 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) for delivery. This works well with e-mail (using filters on the client side) and for error detection (getting e-mail when there is a problem. 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.

Table 6.2. Bcfg2 Report Delivery Mechanisms

Name	Description
www	an XHTML file
rss	an RSS file *(links do not point at real web links, since they may not exist)*
mail	A plaintext e-mail message

Configuration

The `report-configuration.xml` file is the standard file that the **bcfg2-build-reports** executable uses when it is run without any command line arguments. Alternate configuration files, formatted identically, can be used by specifing -c flag. This can be useful for running different types of reports at different intervals. For example:

```
Run this hourly: bcfg2-build-reports -c WebAndRssReport-config.xml
Run this daily: bcfg2-build-reports -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, or by a Python Regular Expression describing a group of hostnames. ".*" is especially helpful to describe all hosts. More information can be found about such Regexes at: http://docs.python.org/lib/re-syntax.html.

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

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

Reporting Quick Start

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

Example 6.1. etc/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>
```

Once configured correctly, just wait for the e-mail or view the outputed html files with a web browser. **bcfg2-build-reports** to recieve your reports immediately, or configure cron to run it perodically. E-mail reports will deliver the appropriate content directly to your mail client, and the html files should be viewable with any web browser. It is suggested those files be accessable via a webserver for convenience to other interested parties.

Examples of the performance and overview reports.