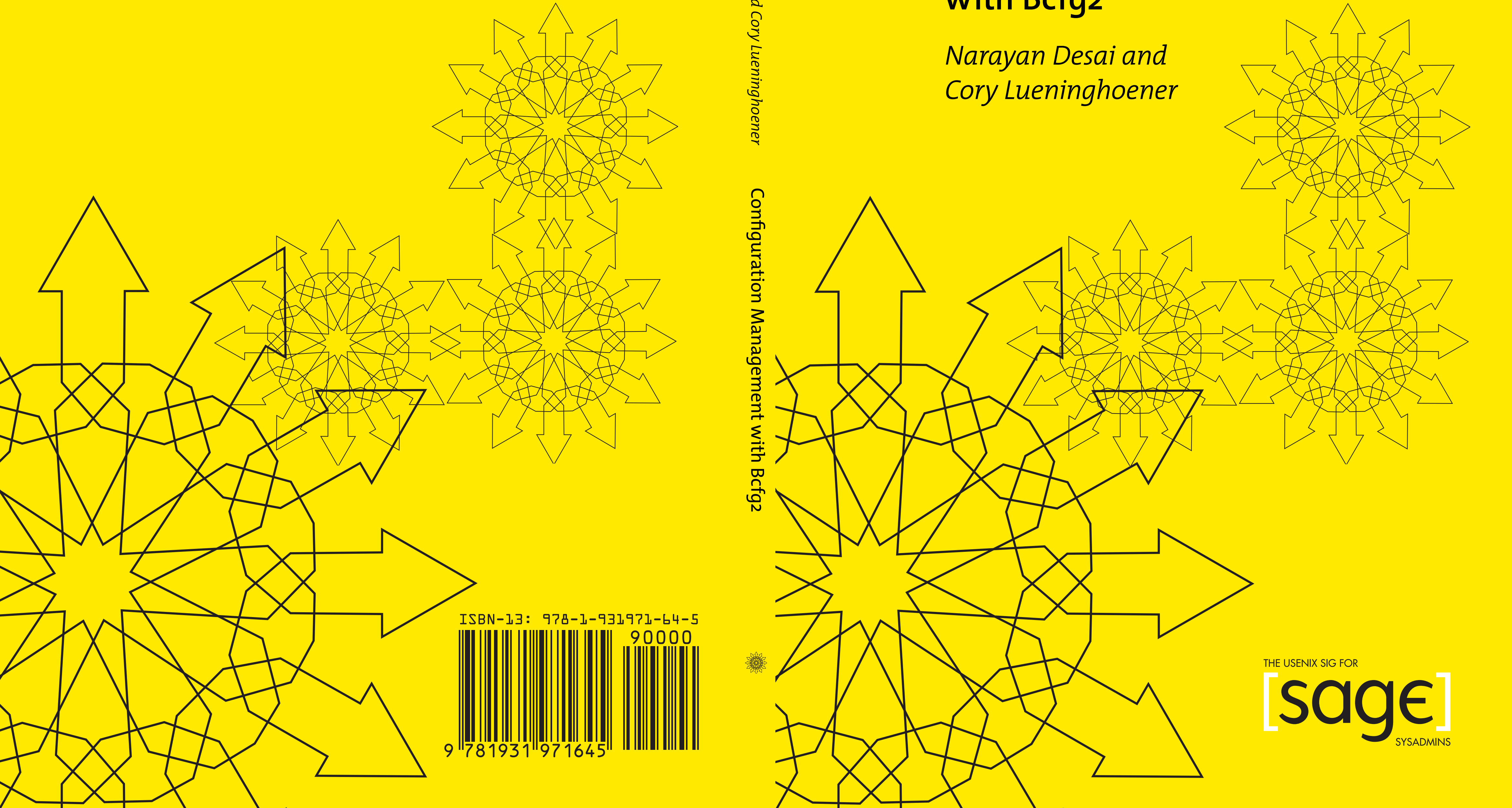
THE USENIX SIG FOR [sage] SYSADMINS

ISBN-13: 978-1-931971-64-5
90000
9 781931 971645

19

Narayan Desai and Cory Lueninghoener

Configuration Management with Bcfg2



19 | *Short Topics in* **System Administration**

Jane-Ellen Long, Series Editor

Configuration Management with Bcfg2

Narayan Desai and Cory Lueninghoener

THE USENIX SIG FOR [sage] SYSADMINS

Booklets in the Series

#19: **Configuration Management with Bcfg2,** by Narayan Desai and Cory Lueninghoener

#18: **Deploying the VMware Infrastructure,** by John Arrasjid, Karthik Balachandran, Daniel Conde, Gary Lamb, and Steve Kaplan

#17: **LCFG: A Practical Tool for System Configuration,** by Paul Anderson

#16: **A System Engineer's Guide to Host Configuration and Maintenance Using Cfengine,** by Mark Burgess and Æleen Frisch

#15: **Internet Postmaster: Duties and Responsibilities,** by Nick Christenson and Brad Knowles

#14: **System Configuration,** by Paul Anderson

#13: **The Sysadmin's Guide to Oracle,** by Ben Rockwood

#12: **Building a Logging Infrastructure,** by Abe Singer and Tina Bird

#11: **Documentation Writing for System Administrators,** by Mark C. Langston

#10: **Budgeting for SysAdmins,** by Adam Moskowitz

#9: **Backups and Recovery,** by W. Curtis Preston and Hal Skelly

#8: **Job Descriptions for System Administrators, Revised and Expanded Edition,** edited by Tina Darmohray

#7: **System and Network Administration for Higher Reliability,** by John Sellens

#6: **A System Administrator's Guide to Auditing,** by Geoff Halprin

#5: **Hiring System Administrators,** by Gretchen Phillips

#4: **Educating and Training System Administrators: A Survey,** by David Kuncicky and Bruce Alan Wynn

#3: **System Security: A Management Perspective,** by David Oppenheimer, David Wagner, and Michele D. Crabb, and edited by Dan Geer

#2: **A Guide to Developing Computing Policy Documents,** edited by Barbara L. Dijker

19 *Short Topics in*
System Administration

Jane-Ellen Long, Series Editor

Configuration Management with Bcfg2

Narayan Desai and Cory Lueninghoener

Published by the USENIX Association
2008

About SAGE

SAGE is a Special Interest Group of the USENIX Association. Its goal is to serve the system administration community by:

- ❖ Offering conferences and training to enhance the technical and managerial capabilities of members of the profession
- ❖ Promoting activities that advance the state of the art or the community
- ❖ Providing tools, information, and services to assist system administrators and their organizations
- ❖ Establishing standards of professional excellence and recognizing those who attain them

SAGE offers its members professional and technical information through a variety of programs. Please see http://www.sage.org for more information.

© Copyright 2008 by the USENIX Association. All rights reserved.

ISBN 978-1-931971-64-5

To purchase additional copies, see http://www.sage.org/pubs/short_topics.html.

The USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA USA 94710

http://www.usenix.org/

First Printing 2008

USENIX is a registered trademark of the USENIX Association.

USENIX acknowledges all trademarks herein.

Contents

Acknowledgments v

1. Configuration and You 1
- 1.1 The Configuration Problem 1
- 1.2 Toward a Solution 3
- 1.3 Deploying Configuration Management 4
- 1.4 How to Read This Booklet 5

2. The Bcfg2 Architecture 6
- 2.1 Design Goals 6
- 2.2 Bcfg2 Architecture 7
- 2.3 Moving Ahead 18

3. Getting Started 19
- 3.1 The Server 19
- 3.2 The Client 20
- 3.3 Making Changes with Bcfg2 21
- 3.4 The Next Steps 24

4. Task Examples 25
- 4.1 Message of the Day 25
- 4.2 An NTP Client 27
- 4.3 Managing the Base Configuration 28
- 4.4 An NTP Client and Server 28
- 4.5 Managing an SSH Infrastructure 29
- 4.6 Using Actions 31

5. Troubleshooting 33
- 5.1 Logging 33
- 5.2 bcfg2-info 35
- 5.3 Client Debugging 36

6. Advanced Task Examples 38
- 6.1 Templates: Setting a Hostname 38
- 6.2 Probes 40
- 6.3 Structure Templates 41
- 6.4 Complex Templates 42

7. The Bcfg2 Reporting System 47
- 7.1 Reporting System Architecture 47
- 7.2 Use Cases 49

8. What Next? 50
- 8.1 Deployment Tips 50
- 8.2 Further Reading 51

Appendix A. Bcfg2 XML Options 53
- A.1 Configuration Processing 53
- A.2 Goal Types 54
- A.3 Plugins 57

Appendix B. Client Identification 60

About the Authors 61

Figures and Tables

Figures

2.1 Information flow in Bcfg2 8
2.2 Basic set of group definitions 14
2.3 A typical top-level Bcfg2 repository 17
4.1 A simple clients.xml 25
4.2 A simple groups.xml 26
4.3 A message-of-the-day Bundle, from Bundler/motd.xml 26
4.4 An NTP Bundle 27
4.5 A simple services.xml 27
4.6 A simple Base file 28
4.7 A group-specific ntp.conf file 29
4.8 An update to groups.xml 29
4.9 An SSH Bundle 30
4.10 A Postfix Bundle using Actions 31
4.11 An Actions file 31
6.1 The HOSTNAME template 38
6.2 A rather silly HOSTNAME template 39
6.3 A HOSTNAME ConfigFile entry 39
6.4 A disk-counting Probe 40
6.5 A template making use of Probe data 40
6.6 An SGenshi template 42
6.7 Some SLES network configuration file names (applies to other distributions as well) 43
6.8 Using the altsrc tag option 43
6.9 A Probe to collect MAC addresses 44
6.10 The Network Structure template 44
6.11 The TCheetah/etc/sysconfig/network/ifcfg-eth-id-MAC template 45
6.12 The network interface configuration file 45
7.1 The Bcfg2 Web reporting front end 48

Tables

6.1 Metadata available to templates 39
A.1 Action attributes 54
A.2 ConfigFile attributes 55
A.3 Package attributes 56
A.4 Instance attributes 56
A.5 Permissions attributes 56
A.6 Directory attributes 57
A.7 Symlink attributes 57
A.8 PackageList attributes 58

Acknowledgments

Narayan would like to thank his wife, Helen, for her support and encouragement throughout this project. Cory would like to thank his wife, Lis, and his cats, Gypsy and Sophie, for their support, companionship while writing, and introduction of typographical errors (respectively).

The authors also thank all of the technical reviewers for their help. Ti Leggett, Rick Bradshaw, and Jason Hedden provided useful organizational suggestions based on an early manuscript. Thomas Delaet and Sol Jerome each performed a complete technical review, resulting in several major improvements. In addition to a technical review, Daniel Clark also tested each of the examples included in this book.

Finally, the Bcfg2 community has greatly refined our understanding of the problem of configuration management and the design of Bcfg2 itself. Scores of intricate discussions have informed the viewpoints presented here.

1. Configuration and You

Configuration is the medium in which system administrators work. Ranging from the most sweeping decisions to minor troubleshooting of software problems, the act of helping people use computers necessitates a stream of decisions that result in the configuration and software infrastructures that each of us uses on a daily basis. Configuration management is a process whereby administrator interactions with configuration are streamlined and simplified. As the scale of infrastructures grows and the complexity of software systems increases, the cost of system administration rises rapidly. Automation techniques, such as configuration management, can reduce these costs by streamlining common tasks. Automation also implements rigorous processes; automated processes are usually performed more consistently than manual ones.

Our intention is to enable you to understand the basics of configuration management and how to use Bcfg2 to achieve common configuration goals. With this booklet, you will be able both to design appropriate configuration processes and policies for your site and to analyze configuration processes to assess their effectiveness.

1.1 The Configuration Problem

In this booklet, we define configuration as the union of software factors that influence the behavior and performance of computer systems. This is a fairly expansive definition of configuration.

When administrators work with software systems, their activities fall into two major categories. The first uses domain-specific software information. These activities require a high level of knowledge about the capabilities of particular software and the semantics of their configuration files. The tasks are common and may involve the configuration of packages such as the Linux kernel, the Apache Web server, or any of myriad other software packages. This area involves understanding the ways to properly activate and use given software packages.

The second category is what we consider to be configuration. These tasks involve building coherent services from the software components described earlier. Here, high-level requirements become relevant; these may be technical or nontechnical. Operational requirements figure into these systems as well; correctness, security, and robustness are frequent goals of the configuration process.

Activities in both of these categories are required in order to achieve results; the second is clearly dependent on the first. At the same time, we find that the second category poses a much larger and more persistent set of problems for administrators. Hence, these activities are the explicit targets of configuration management, and specifically the architecture of Bcfg2.

1.1.1 Interacting with Configuration

Interacting with configuration is unwieldy for several reasons. Traditional configuration is distributed, potentially existing on every device in a network. For example, firewall configurations on network devices contribute to the operational state of services as much as any configuration located on the server itself. Configuration itself is frequently disorganized and inconsistent, making it hard to troubleshoot systems.

Moreover, interaction with traditional configuration scales poorly along several different axes. As the number of devices grows, so does the sheer quantity of configuration. The amount of configuration diversity present also increases the cost of configuration, as it reduces the amount of commonality between systems. Additionally, administrator count causes scalability problems in a counterintuitive fashion. As more administrators contribute to the operation of systems, more coordination is required. Traditional configuration is poorly suited to aid in this coordination.

1.1.2 The Configuration Process

Configuration itself isn't a monolithic process. Administrators interact with configuration in a number of different ways:

- ❖ Creation: Administrators create new configuration whenever new services or systems are deployed.
- ❖ Modification: Administrators modify existing configuration as service requirements change or as new software is released. Security updates frequently provide an urgent example of this category.
- ❖ Analysis: Administrators need to analyze existing configuration in order to acclimate themselves to unfamiliar surroundings. Also, administrators frequently need to reacquaint themselves with configurations during troubleshooting.
- ❖ Validation: Administrators need to convince themselves or others that the overall configuration adheres to organizational policies.

Each of these tasks can be greatly eased through new configuration-based functions. For example, configuration creation becomes considerably simpler if existing configuration can be reused. Likewise, modifications can be automated with a machine-consumable definition of configuration state. Modifications can be further streamlined if the specification format is one that can be generated robustly; this allows the construction of programs that generate configuration updates in response to external events.

Analysis poses one of the most pressing issues facing organizations. The combination of staff turnover with a lack of sufficient cross-training and up-to-date documentation requires administrators to work in situations where they need to discover how systems are configured. This requirement leads to a large spin-up time for new employees and a higher time to solution when troubleshooting.

Exacerbating this situation is the need for configuration validation by external entities, be they management, funding sources, or regulators. Anyone who has been through an audit understands the sheer volume of data that must be collected and presented. Traditional configurations are simply difficult to audit.

Together, all of these issues suggest that a different approach is needed.

1.2 Toward a Solution

Manipulation of traditional configuration is so fault-prone that an alternative approach is required. Usually, configuration systems focus on easing the performance of configuration changes, without addressing the other areas of the configuration problem. Much more expansive than those typically held by configuration management systems, the Bcfg2 approach is to build a useful representation of configuration goals that can be correlated with current configuration state. This representation needs to be accurate, compact, verifiable, and centralized. Moreover, these goals must be reconcilable with the current state of systems in a flexible, efficient, and intuitive manner. These requirements are much more expansive than those of most configuration management systems.

Following the requirements that form the basis for Bcfg2's architecture and approach, we have attempted to streamline the four aspects of configuration described in the previous section.

Bcfg2's model stores two bodies of information: the configuration state of managed systems and a series of configuration goals, describing the desired configuration state. These goals have three important characteristics: they are verifiable, installable, and discoverable. The definition of goals makes them directly comparable with current system state. Whenever this comparison finds inconsistencies, reconciliation is needed.

Depending on the environment, inconsistencies can imply one of two situations. Either new goals have been specified and should be enforced on clients, or new configuration has been introduced on managed systems and should be reflected in the goals. In many cases, inconsistencies can be resolved by convention (i.e., "No one should make configuration changes directly on servers"), but exceptions frequently occur for legitimate reasons. Therefore, Bcfg2 supports bidirectional flow of configuration data between the manager and managed systems. This is a key, and unique, feature of Bcfg2.

The isolation of goals from state also makes the configuration process transparent. This added information makes it possible to make more effective and informed decisions about management policy.

1.2.1 What Is Bcfg2?

Bcfg2 provides a mechanism for describing system configurations for large, heterogeneous environments of systems. More important, it collects data to streamline the configuration discovery process. Most environments are pre-existing; the introduction of a new configuration tool does not reduce the importance of existing infrastructure. Indeed, it is not always feasible to rebuild all client systems in order to *properly* manage them.

As a tool, Bcfg2 has been designed to work in a noninvasive manner. Although it can be used as a proscriptive and assertive tool, with complete control over client configurations directed from a central location, it can also be used to selectively manage parts of client configurations and track configuration changes performed on clients. This architecture imposes no deployment strategy on administrators, leaving them free to deploy Bcfg2 in the role that makes sense in a particular environment. In pre-existing environments, this means busy administrators can start by managing their most important configuration elements while working their way toward a more complete configuration state.

Bcfg2 is designed to help administrators get their machines under configuration management quickly. It doesn't require complex modeling and administration paradigm shifts to use effectively, which means it is very accessible to all administrators and environments. Used correctly, it offers administrators benefits that become clear very early in the deployment process.

1.3 Deploying Configuration Management

Effective use of software requires understanding the software itself and determining a strategy that is appropriate for a given situation. In this, Bcfg2 is not an exception.

Most system administrators do not have the luxury of deploying a new infrastructure from scratch. The cost of rebuilding all currently existing services and resources is often prohibitively high. Given this limitation, configuration management processes must be layered on top of existing infrastructure, which may be disorganized, pathological, or not well understood.

The deployment of configuration management processes has two main benefits. The first is simple: decreasing the cost of the configuration process. The driving need for this improvement is clear: administrators are expected to manage increasing numbers of systems with fixed or decreasing staffing. The second benefit is a side effect of performing the first: the resulting configuration specification provides a central point of information about systems, their roles, and their required configurations.

Centralized configuration knowledge provides a number of benefits to administrators, particularly in groups. It provides a shared repository of institutional knowledge that administrators can readily use. It functions as a configuration plan of record for systems. Moreover, it injects visibility into the configuration process. Previously, this process was performed in relative darkness. This new information can be used both to assess the performance of the configuration process and to better understand the costs associated with system management.

All of these benefits combine to provide a safety mechanism for system administrators. Configuration management is a potent tool for positive change in system administration environments. However, it has an unparalleled destructive capacity. For this reason, administrators must be provided sufficient information to make informed decisions. Using this information effectively can minimize the danger of having administrators with an incomplete understanding of deployed configurations.

In view of this setting, configuration management processes must first focus on safety, before any other considerations. By safety, we mean that both the configuration management system and processes are designed to ease configuration discovery and build robust, configuration-sensitive control processes. Bcfg2 has been designed to provide these facilities; they are discussed in Section 3.3.

1.4 How to Read This Booklet

This booklet has eight chapters. They are topically split between conceptual overviews and hands-on examples. Chapter 2 provides a high-level architectural overview of Bcfg2. We describe each of the major components of Bcfg2 in turn, noting their roles and interaction points, and highlighting important design features. This chapter provides the big picture; readers who wish to begin by understanding the high-level architecture should start here.

Alternatively, readers wanting to dive in can begin with Chapter 3. This chapter demonstrates how the architecture is used in practice. Here, we provide concrete details showing how Bcfg2 works and how to use it effectively.

Chapter 4 describes how Bcfg2 can be used to accomplish a series of common tasks. Each task highlights several important aspects of working with Bcfg2. Chapter 5 explains the mechanisms that can be used to troubleshoot Bcfg2. Chapter 6 provides an advanced set of examples that use a wide range of Bcfg2 features to achieve sophisticated goals. Chapter 7 offers an overview of how Bcfg2 records system-wide status information and detects any deviation from, or inability to reach, the desired state after the system has been modified. Finally, Chapter 8 provides tips on how to deploy Bcfg2 after configuration policies have been put in place and configuration tasks have been fully automated, along with some additional resources.

Detailed information about low-level formats and processes is provided in two appendices; Appendix A describes the file formats used by the Bcfg2 server, while Appendix B describes the methods used for client identification and authentication.

2. The Bcfg2 Architecture

In this chapter we discuss the design and architecture of Bcfg2. We begin with an overview of the goals, moving on from there to the system architecture. We then explain the implementation of Bcfg2, describing each component of the system in detail. This includes details on the operation of the Bcfg2 server, client, plugins, and reporting system.

This chapter is most useful for those who want to fully understand the design and operation of the entire Bcfg2 system. Users who want to dive straight in and start installing and using the system can skim through this chapter to gain an overall understanding of the system and then move on to the next chapter, which begins by installing the client and server. We do recommend fully reading through this chapter at some point, though, because it yields a much better understanding of the entire system than task-specific use of the tool can provide.

2.1 Design Goals

As we have emphasized, traditional configuration has a number of drawbacks. This realization drove the design of Bcfg2. Our main concerns were:

- Uncertainty: Traditional configuration is distributed and difficult to examine. In some cases it is not wholly available at any point in time. We became worried about our loosely coordinated configuration processes because we didn't have sufficient visibility to ensure that they were working properly.
- Inefficiency: The distributed nature of traditional configuration makes it difficult to efficiently deploy similar changes to large numbers of clients. The deployment of configuration changes was likewise costly and error-prone.
- Inconsistency: The distributed nature of traditional configuration, in conjunction with ad hoc administration, introduces a large number of unintended differences into configurations. These small differences, while usually harmless, caused behavior differences that were frequently time-consuming to repair.
- Disorganization: Traditional configuration is unstructured, which makes it hard to analyze and troubleshoot. Any commonality must be divined from manual examination. Troubleshooting frequently took longer than it would have in a more organized environment.
- Poor scalability: Administrators are facing scalability problems in every direction. Not only is the ratio of systems to administrator growing, but the amount of configuration diversity present in common networks is quickly growing as well. Moreover, in the lucky cases where staffing is increased, organizational problems stemming from institutional knowledge become an issue. Traditional configuration is poorly structured to ease the burden of these increases.

2.1.1 Operational Goals

While these high-level concerns motivated many aspects of the Bcfg2 architecture, several environmental factors and operational goals fed into the process as well.

We work in a computer science research environment. It consists of a typical server infrastructure and workstation environment, several large high-performance computing clusters, and a variety of experimental systems. From the configuration perspective, it has several important characteristics. We support a large number of resources for computer science systems research; in some of these cases, users share root privileges with the administrative staff on a transient basis. In other cases, where users need to build services as a part of their research, root privileges must be permanently shared. And because these services have external users, standard robustness, serviceability, and security issues apply. Sharing responsibility for user-visible services made us keenly aware of many organization and consistency issues.

These considerations led to a set of functional requirements for Bcfg2:

- ❖ Provide a coordination point for administration on systems with shared root access.
- ❖ Maintain current administration methods.
- ❖ Accelerate the creation of *one-off* configurations.
- ❖ Scale to large client counts.
- ❖ Collect reliable and up-to-date information about the configuration states of clients, including the results of unexpected administration activities.
- ❖ Improve efficiency of common configuration operations.
- ❖ Enable appropriate management of legacy systems.

In addition to these functional requirements, usability requirements were featured prominently in early discussions. Usability discussions tend to be difficult because of the lack of consensus about administrative techniques. Eventually, we decided that administrator comfort is largely a matter of personal choice. Hence, Bcfg2 needed to support as wide a range of administration methodologies as possible, including completely manual administration.

2.2 Bcfg2 Architecture

The Bcfg2 architecture is built to explicitly model administrator goals, system configuration state, and the process that merges them. Administrator goals describe the state that administrators want system configurations to be in. At any given time, these goals may or may not be satisfied. It is Bcfg2's job to enumerate these goals, compare them with the current state, and determine which goals have not been satisfied. Information about these inconsistencies is gathered and stored on a central Bcfg2 server. When one is presented with an inconsistency, several courses of action are possible. In one case, the goal state can assert authority over the current state, resulting in a reconfiguration operation on a client. This is a frequent outcome. In some cases, however, it is preferable for the current client state to win out over goals. This occurs in a variety of distributed administration situations. In this case, information about the current state of the client must be integrated into the explicit goals.

This analysis of requirements led us to Bcfg2's software architecture. Early in the process, we decided that the system needed to be centralized in order to allow administrators to effectively manage large numbers of clients in a uniform manner. All control flows from the central server out to clients.

Bcfg2 uses a client-server architecture. The bcfg2-server process runs on a central system. It uses a collection of configuration rules to render client-specific configuration goals on demand. By the time the repository is rendered into a client configuration specification, all ambiguity has been removed; no client-side processing is required in order to determine the desired configuration. These goals are passed to clients, where they are validated against local state and reconfiguration operations can be performed. Once all operations are complete, the client uploads transaction and state information into the reporting system, where it can readily be examined and analyzed. Figure 2.1 shows the flow of information through the system.

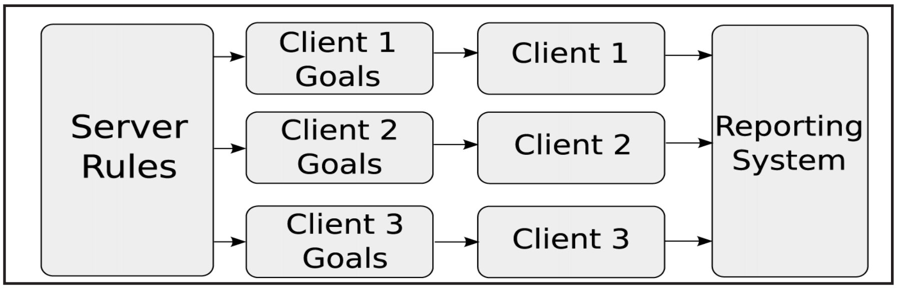


Figure 2.1: Information flow in Bcfg2

Bcfg2 has been designed to perform complicated processing centrally. Central configuration processing contains complicated processes on the server, where they can be readily examined and supervised. When goals pass from the server to clients, they are detailed and literal; this strategy minimizes the amount of processing that clients must perform.

2.2.1 Configuration Goals

Configuration goals provide a snapshot of the desired state of client systems at a point in time. Goals are collections of configuration entries that describe particular aspects of configuration state. These entries correspond to all of the common configuration entities with which system administrators frequently interact. Packages, configuration files, services, and a variety of POSIX filesystem objects can all be represented by using Bcfg2 configuration entries. These entries are described in detail in Appendix A. They have been chosen to have common, simple definitions, shared by nearly everyone. Entries describe the desired end-state of the configuration item, as opposed to the process needed to implement them. Goal entries include all information needed to both verify and install them on clients. For example, package entries include name and version information, as well as source information, if needed. Configuration file entries contain file contents and all ownership information. Service entries contain status information. All entries have idempotent semantics, which means they produce stable results upon repeated verification and installation.

Goals also contain grouping data for entries. These groupings, called Bundles, describe the relationships between entries. Bundles can be used to relate configuration file entries to the software packages they affect. Similarly, services can be associated with these groupings as well. Bundle associations cause two changes in semantics. Entries contained in the same Bundle are validated collectively; that is, all entries get reverified whenever any entry is modified. Also, services included in a Bundle are restarted whenever any member entry is modified.

By convention, Bcfg2 assumes goals are comprehensive. This means that goals describe all aspects of software configuration on the client. If a configuration entity is not included in goals, then it is assumed to be unintended. This assumption, while not mandating any client behavior, allows robust detection of subtle goal mismatches. Configuration goals can range greatly in size; the smallest complete configuration goals typically contain around two hundred entries, while larger desktop configurations may contain three thousand or more.

The semantics of configuration goals provide the foundation for Bcfg2's architecture. Not only can individual entries be constructed from the inspection of client system state, but the relationships between entries can be discovered. This ability allows Bcfg2 to be much more tolerant of user error and to streamline recovery in such cases. Moreover, because these types are functionally universal, the client can pass to the server data that can be directly used to refine and correct the server-side configuration specification. The model is unique to Bcfg2 among configuration tools. It allows Bcfg2 to be useful in more situations than traditional configuration management tools.

2.2.2 The Bcfg2 Client

The primary consumer of configuration goals is the Bcfg2 client. It is responsible for downloading configuration goals from the server, comparing the local system configuration state to goals, determining and performing appropriate configuration actions, and collecting summary information to send to the reporting system.

Transparency and robustness are the two main goals of the Bcfg2 client implementation. Because the Bcfg2 client is the only distributed part of the system, robustness is the single most important characteristic. To this end, configuration goals are completely specified before they are sent to the client; this approach minimizes the amount of processing that must be performed on the client side. Transparency is also critical, but in a different way. Because the Bcfg2 client can perform all system reconfigurations, administrators need to understand its operation during all manual interactions.

Three main aspects of the Bcfg2 client warrant description here: the basic operation of the Bcfg2 client; tool drivers, which provide all of the change performance functionality for the client; and mechanisms used to trigger client reconfigurations. Each of these topics is described in some detail below.

BASIC OPERATION

As we mentioned earlier, the Bcfg2 client interacts with the server, using the resulting information to coordinate its operations. Seven main actions are taken by the Bcfg2 client.

The client: (1) downloads and executes probes, uploading the results to the server; (2) downloads its configuration specification from the server; (3) loads and instantiates tool drivers, giving the client the ability to probe and alter the configuration state of the local system; (4) collects an inventory and compares it with the configuration goals provided by the server, resulting in a set of candidate configuration operations; (5) decides on appropriate actions; (6) takes those actions; and (7) uploads a detailed description of its state and actions to the reporting system. We describe here each of these steps in detail:

- Execute probes: The client downloads a set of probes to be executed locally. These probes allow the detection of client characteristics. The results of these probes are uploaded to the server for use in the generation of the client's goals. For example, client video hardware can be detected, allowing the automatic generation of a tailored X configuration for clients. While this data could be tracked on the server, it is only a reflection of the state of the client and could easily become inaccurate. The use of a client-side probe enables the use of canonical data at all times.
- Download configuration goals: The client downloads its configuration goals from the Bcfg2 server. Goals are completely literal at this point; the client does not need to do any further processing in order to either verify or install them.
- Load tool drivers: Next, the Bcfg2 client attempts to load and instantiate all tool drivers. Each driver has a list of prerequisites for its operation and will load only if they are met. This means that drivers are automatically available when they will operate. For example, if RPM is installed on a Debian system, the RPMng driver will load properly. If running in verbose mode, the Bcfg2 client will display a list of successfully activated drivers. A complete list of tool drivers is included below.
- Inventory local state: Once the Bcfg2 client has instantiated all tool drivers and has a copy of the configuration goals, it compares its local state to the state described in the specification. This process is performed in two parts. First, the state of each entry included in the configuration specification is validated. This step produces a list of incorrect entries. Second, client tool drivers use heuristics to discover configuration entries that are not included in the configuration specification. These entries, called extra entries, can cause as much of a problem as any incorrect entry. This stage detects how well a system conforms to the goals provided by the server and what unspecified configuration exists on it.
- Decide which changes to apply: After the inventory stage, the Bcfg2 client has produced a list of incorrect and extra entries. Depending on its mode, it will decide to correct all, some, or none of these items. A list of available modes and their behavior is included below.
  - Regular mode: Attempts to correct all incorrect entries. May be combined with removal and central decision modes.
  - Dry-run mode (`-n`): Performs no operations of any sort. Will not make any changes to the client, but will send updated report data back to the server.
  - Interactive mode (`-I`): Prompts the user for each incorrect or extra entry, providing detailed information about the proposed operation. May be combined with removal mode.

    - Removal mode (-r): Removes extra entries of the type specified, either all or packages. Controls only operations pertaining to extra entries, and may be combined with normal or interactive modes.
- Apply configuration changes: The completed decision phase provides the client with a comprehensive list of operations to undertake. It then performs all entry removal operations followed by all entry modification or installation operations. These steps are repeated while there are still pending (or failed) operations; thus, the number of pending operations decreases per pass. This allows the client to remove packages that conflict with packages that need to be installed.
- Report on final state: Once all operations that can be performed successfully have finished, the client uploads a set of configuration statistics to the server. This data includes a list of all still-incorrect entries, all extra entries, and all entries modified during execution. This information forms the basis for the Bcfg2 reporting system, described in Section 2.2.3.

TOOL DRIVERS

The activities described in the previous section are implemented as the core logic of the Bcfg2 client. All verification and installation operations are implemented through a plugin mechanism. These tool drivers provide the glue needed to interact with underlying system management tools. Each of these drivers is relatively simple; the smallest is fewer than 50 lines of code. The following is a list of the drivers included with Bcfg2-0.9.6:

- Action: The Action driver provides logic pertaining to pre- and post-installation actions. This driver is also available on all systems.
- APT: The APT driver provides integration with the APT package manager. It is available on systems where it is used, typically Debian, Ubuntu, and Nexenta systems.
- Blast: The Blast driver provides integration with Blastwave packages and is used on Solaris.
- ChkConfig:The ChkConfig driver provides integration with the chkconfig service management system used on Red Hat–like systems.
- DebInit: The DebInit driver provides integration with the service management system used on Debian and Ubuntu systems.
- Encap: The Encap driver provides integration with the Encap packaging system.
- FreeBSDPackage: The FreeBSDPackage driver provides integration with the FreeBSD packaging system.
- Portage: The Portage driver provides integration with the Portage package manager used on Gentoo systems.
- POSIX: The POSIX driver provides logic pertaining to POSIX filesystem entries, handling ConfigFile, SymLink, and Directory entries. This driver is available on all systems.

- RcUpdate: The RcUpdate driver provides integration with the service management system used on Gentoo systems.
- RPMng: The RPMng driver provides the ability to interact with the RPM package system. It handles Package entries of type rpm. It is typically used on systems that use RPM as their native packaging system, but it is available on others if RPM is installed.
- SMF: The SMF driver provides the ability to interact with Solaris's Service Management Facility. It is available on Solaris and Nexenta systems.
- SYSV: The SYSV driver provides the ability to interact with Solaris SysV packages. This driver is used only on Solaris systems.
- YUMng: The YUMng driver provides analogous functionality to RPMng, but it uses the yum utility to install packages. It is typically available on Red Hat and Fedora systems.

SCHEDULING CLIENT EXECUTION

The Bcfg2 client is typically triggered in one of four ways, depending on site requirements. First, execution at boot-time is used to apply updates that have occurred while a host is offline. This mechanism is used nearly universally. The second and also nearly universal mechanism used to run the Bcfg2 client is cron. We recommend that each Bcfg2 client run at least once per day, but some sites run it hourly.

High-performance computing clusters have a different set of constraints because transient tasks can cause serious performance problems for applications. In these cases, sites typically run the Bcfg2 client from job prologue or epilogue scripts. This strategy ensures that the Bcfg2 client is run only while nodes are idle.

Finally, the Bcfg2 client can run as a daemon that waits for connection from the Bcfg2 server in order to initiate client executions. This mechanism can be used as frequently or infrequently as circumstances dictate.

Each of these mechanisms is configured in a common fashion. Operation of each mechanism is controlled by the file /etc/default/bcfg2. Each mode can be individually enabled and configured, allowing the use of distinct client options in different modes.

2.2.3 Reporting System

The reporting system acts as the primary user interface into the configuration management process implemented by Bcfg2. From this view, administrators can easily determine the overall configuration state of all managed systems. The reporting system offers an integrated view of high-level summary statistics that can be used to drill down to detailed per-entry information about misconfigurations or extra entries. In addition to summary and detailed information, the reporting system provides correlations between detailed misconfigurations. This allows additional insight into the effectiveness of the configuration process and class information in cases where it is working poorly.

The reporting system is structured as a database with Web and command-line front ends. Both of these interfaces give access to the same data, for different purposes. The

Web interface is useful for human consumption, while the command-line front ends are more useful for scripting.

The reporting system presents a view of the state of the configuration system as a whole. The new idea here is that the reporting system supplies concrete metrics of how well configuration management is being used. Hence, administrators can see how well configuration management corresponds to the current state of systems. Any differences are a sign of a problem; either the configuration specification is incorrect, or the client hasn't applied updates properly. These differences also represent what would happen if a system was rebuilt directly from Bcfg2, for example in the event of a system disk failure.

2.2.4 The Bcfg2 Server

We have described configuration goals, including their contents and uses, and have presented the operation of the Bcfg2 client and reporting system. The only remaining piece of the Bcfg2 system to be discussed is the server. The Bcfg2 server acts as a repository of configuration rules. When a client requests its configuration, the Bcfg2 server renders these rules into a set of configuration goals for that client, with all ambiguity removed.

The Bcfg2 server is implemented as a Python daemon that speaks XML-RPC over secure HTTP. It runs on a central system (or systems) and is frequently co-located with the reporting system. The server process, called `bcfg2-server`, caches all rules in memory and uses file monitoring to ensure cache coherency. It stores all rules in a central repository, typically /var/lib/bcfg2.

Many of the design goals of Bcfg2 center on the ability to effectively interact with a convenient and useful representation of configuration. The result of this process is the configuration rules used by the server. These rules have been designed to be compact, easily reusable, and extensible. Each rule is associated with a particular group of clients. In this section, we will describe the role and function of client metadata and the structure of configuration rules. After we have explained these building blocks, we will describe the process whereby the server generates a client's configuration goals. Finally, we will show how to use the Bcfg2 repository.

CLIENT METADATA

At a high level, server-side configuration rules describe a configuration goal for a given set of clients. This set can consist of a single client, a subset of clients, or all clients. Configuration rules use client metadata to scope configuration rules. In effect, client metadata is used to describe which rules apply to which clients. When a client connects to the Bcfg2 server, the server daemon identifies and authenticates it. This process is described in detail in Appendix B.

Each client has a unique metadata instance. This metadata comprises three pieces of data. The first is the client's name, usually the client's hostname. The second piece of data is a set of group memberships. A client can be a member of an arbitrary number of groups. These groups frequently correspond to some aspect of configuration similarity between clients. For example, client architecture and base operating system versions are frequently represented by using groups. Client roles are also frequently represented with groups. Because groups can contain as large or small a set of clients as needed for a configuration rule, the number of configuration rules is minimized; the same configuration rule usually does not need to be specified more than once.

Many groups describe independent functional characteristics of clients. These groups are similar to virtual base classes in object-oriented systems. They convey some amount of information about commonality about clients but do not contain enough information to be directly instantiated. A concrete example is a group corresponding to the X86_64 architecture. While this detail can imply substantial configuration similarity among a large number of clients, this group does not provide enough data to build a complete set of client goals; after all, an X86_64 Web server is much different from an X86_64 desktop. Indeed, clients sharing an architecture may not even share an operating system.

Another use of groups is to aggregate clients into classes based on like software configuration. For example, clients in the desktop group might include additional software and services for X Windows. These groups may still be incomplete; that is, they describe only certain aspects of configuration, as opposed to a complete view of client configuration.

The final use of groups is to compose these feature groups into a complete set of aspects. These groups, called *profiles,* contain a large number of groups and combine architecture, operating system version, and system role-based groups to provide a complete view of all aspects of system configuration. Profiles are special groups; a client can be set as an instance of a profile. Profiles are the final element of client metadata.

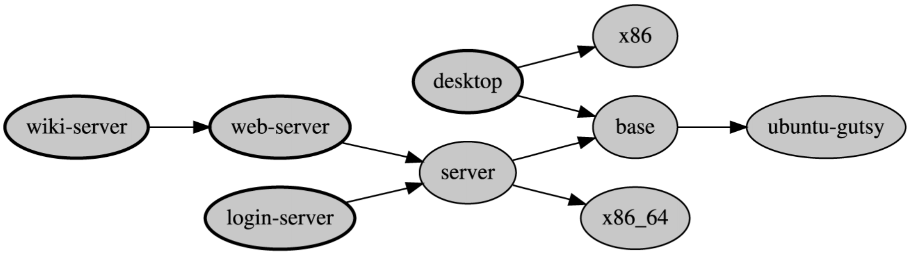


Figure 2.2: Basic set of group definitions

Figure 2.2 shows a diagram of a simple Bcfg2 group hierarchy. This diagram was generated by the `bcfg2-admin viz` command. This example contains several feature groups and several composition groups, as well as four profiles. The four profiles—desktop, login-server, web-server, and wiki-server—are highlighted with a bold outline in the diagram. In addition to the profiles, which are clearly composition groups, the groups base and server also compose lower-level groups. The terminal groups x86, x86_64, and ubuntu-gutsy are basic feature groups. This setup, even in the absence of configuration rules, describes several patterns of configuration in this fictional network. The relationship between the wiki-server and web-server groups enables the reuse of configuration rules. Instances of the wiki-server are identical to instances of the web-server group, with the addition of a membership in the wiki-server group. Rules associated with this group differentiate wiki-server instances from web-server instances:

- ❖ All client systems are members of the ubuntu-gutsy group.
- ❖ The network consists of a combination of x86 and x86_64 clients.
- ❖ All servers are members of the x86_64 group.
- ❖ All desktops are members of the x86 group.
- ❖ All configuration rules intended for all clients apply to the base group.
- ❖ All configuration rules applicable to servers, regardless of type, are associated with the server group.
- ❖ All configuration rules common to all Web servers, regardless of the eventual application, are associated with the web-server group.

This example is extremely simplistic. Typical environments use larger numbers of groups, particularly as Bcfg2 grows to manage most of the configuration on clients.

Group diagrams are quite useful in several cases, because they provide a quick overview of the configuration structure of the whole network. We have found them invaluable for training new personnel. They are also useful when administrators are called upon to work in unfamiliar areas.

CONFIGURATION RULES

Rules contain three important pieces of information: (1) the type of contribution that the rule makes, either entry assertion or entry description; (2) the client group that the rule applies to, whether it's an individual client or all clients; and (3) the priority of the rule. When a client is a member of multiple groups, each of which has a rule pertaining to the same entry, priorities are used to choose which rule should be used. These three pieces of data provide the ability to specify configuration goals for all clients in a compact and efficient fashion.

Configuration rules can contribute to client configuration goals in two ways. First, they can assert that a configuration entry should appear in a client's goals. Second, they can assert that configuration goals should include a particular version of an entry. In concrete terms, one rule may state that client goals should include a goal for the configuration file /etc/passwd, while another may state that members of the web-server group should get a particular version of /etc/passwd, with the associated file contents and metadata.

The scope of a rule is determined through a Boolean expression, with group memberships as terms. For example, a group may apply to clients in groups a and b, either group a or b, or group a and not b.

Rule priorities are another mechanism that ensures the compactness of configuration rules. Using rule priorities, one can specify a rule that applies to all clients and a second, more specific rule that applies to a particular group at a higher priority. The result of this arrangement is that the low-priority default rule applies in all cases where there is not a more specific, and hence higher-priority, version of the entry.

When the Bcfg2 server builds client configuration goals, it first processes all rules that assert entry presence. The result of this step is a list of entries that form the basis for the goals. Each of these entries implies only the presence of the entry, no more. At this point, the server iterates through each entry in the configuration goals and, for each entry, determines which content rules apply to the client. At this stage, rule priorities are used to choose the highest-priority content rule for the entry; this rule is used to construct the full goal entry. After this process is completed, each entry included in the client's configuration goals has all of the information the client needs to verify its state and take corrective action.

SERVER PLUGINS

Despite the abstract simplicity of the rule model, building a good user interface to rules is difficult. Different types of entries have different management tradeoffs and pain points. Similarly, the management of some rules has underlying structure that can lead to simpler user interfaces. Recognizing the difficulty in designing a single language to describe all configuration tasks, we have implemented the Bcfg2 server around a plugin architecture.

Each plugin is able to provide both assertion and description rules. The use of each plugin for these purposes is governed by the structure and generator entries in /etc/bcfg2.conf. Plugins included in the structures line can assert entry membership in client configuration goals. Plugins included in the generators line can provide content rules for entries. Any plugin can be listed in either line, or both, but each of the plugins included with the base distribution of Bcfg2 functions solely in one category or the other. Also, content rules can be provided only by a single plugin per goal entry. Thus, only one plugin can provide content rules for the configuration file /etc/passwd at any time.

A series of plugins is provided with Bcfg2. The following list comprises structure plugins, that is, plugins that provide assertion rules:

- Bundler: The Bundler plugin provides a mechanism that can be used to describe goal entry Bundles. Each Bundle includes a series of global and conditional entries. When the server generates client goals, the Bundler returns several Bundles based on the client's group metadata. Each Bundle includes a set of interdependent entries. Frequently, Bundles correspond to services. Bundles need to be explicitly enabled by including a Bundle reference inside a group definition in Metadata/groups.xml.
- Base: Base provides a mechanism for describing unrelated entries. Because internal group differentiation is all that is needed to describe the mapping of entries to clients, no external assignment of base groups to clients is needed. The file format used by the base plugin includes lists of goal entries enclosed within group predicates. All entries included within a predicate matching the client will be included as goal entries.
- SGenshi: SGenshi provides a templating mechanism for dependent and independent groups of goal entries. The input template is processed in conjunction with client metadata to produce XML similar to that used by Bundler and Base. While SGenshi is a proper functional superset of Bundler and Base, it is

more complicated to use. It is frequently used in cases where entry assertion adheres to more sophisticated patterns than can easily be represented with Bundler and Base. Several examples of SGenshi use are included in Chapters 4 and 6.

- Generators: Generator plugins provide content rules for goal entries. Generators tend to be purpose-built; that is, they are designed for either the management of a particular type of entry or a particular range of entry instances related to a single administrative task. Most generators are generically useful when managing particular types of configuration entries; however, one task-centric generator is included with the base distribution of Bcfg2.
- Rules: The Rules plugin corresponds to the abstract ideal of configuration rules. Each rule used by this plugin is labeled with a priority and scope and includes a set of entry contents. Service and POSIX entries are typically managed by using this plugin.
- Pkgmgr: The Pkgmgr plugin is a slight refinement of the Rules plugin, tailored to handle features needed for handling package version and architecture data. Pkgmgr can handle only Package entries.
- Cfg: The Cfg plugin is purpose-built to manage rules governing configuration file contents. Each file managed by Cfg has a discrete directory, in which all versions of that configuration file are stored, including group and priority labels.
- SSHbase: The SSHbase plugin is the only task-driven plugin included with Bcfg2. Its sole function is to manage client ssh keys and build a correct ssh_known_hosts file. This plugin is able to retain ssh keys across client rebuilds, revoke ssh keys from ssh_known_hosts files network-wide, and ensure the correct and complete generation of ssh_known_hosts files, including proper localhost lines.
- TCheetah: TCheetah is a plugin that provides a mechanism for generating configuration file contents based on a combination of client metadata and static templates. It uses the Cheetah templating system. It is useful for building dynamic configurations based on sophisticated rules or the interpolation of client probes.
- TGenshi: TGenshi, like TCheetah, is a configuration file templating mechanism. TGenshi is based on the Genshi templating language. It has roughly the same uses, features, and interface as TCheetah.

```
Metadata/     Rules/        Svcmgr/       etc/
Pkgmgr/       SGenshi/      TCheetah/
Probes/       SSHbase/      TGenshi/
```

Figure 2.3: A typical top-level Bcfg2 repository

As we mentioned previously, the Bcfg2 server has a single repository that contains all data needed to operate. This repository is usually located in /var/lib/bcfg2. Figure 2.3 shows a top-level view of a typical repository. In the repository, each plugin controls the subdirectory with the same name. The Bundler plugin controls the /var/lib/bcfg2/

Bundler repository subdirectory, and so forth. Three of these directories are special: the Metadata and Probes directories are used by the Metadata module, and the etc directory is used by the reporting system. Each plugin determines the semantics of the files included in its subdirectory, and each of these directories will be examined in more detail as they come up in later chapters.

2.3 Moving Ahead

In this chapter we've taken a high-level look at all of the pieces that make up the Bcfg2 system, which is important for understanding the mindset in which the tool was written. In the next chapter we will look at actually installing the Bcfg2 server and client, setting up a simple repository, and beginning to perform real work on real machines. The chapters after that will then look at numerous real-world tasks that have been performed with Bcfg2 in a cookbook-like style that should help both the new and seasoned configuration management user become productive quickly.

3. Getting Started

3.1 The Server

The first step to getting started with Bcfg2 is to install the server package. While the Bcfg2 server is available as a default package for several GNU/Linux distributions, and packages for several other distributions and operating systems can be found on the Bcfg2 Web site, it can be installed on any operating system that can support its prerequisites. In general, it is pretty certain to work on any modern UNIX-like operating system. See the packages and documentation at http://www.bcfg2.org for more details.

The examples in this section were conducted on a system running Ubuntu Hardy Heron, using Bcfg2-0.9.6. Previous versions of Bcfg2 may not work precisely with this series of steps; at least one feature needed is not available from the command line (-F, used in Section 3.2.1), while several other processes (including bcfg2-admin init) have been refined in 0.9.6.

3.1.1 Initial Setup

Once the server software is installed, it is time to set up an initial repository. The first step is to run the Bcfg2 initialization script on the server machine:

```
> bcfg2-admin init
```

This interactive command will set up a skeleton repository, build an initial /etc/bcfg2.conf file, set up an SSL certificate, and perform any other actions needed to bootstrap the Bcfg2 server. This script will prompt for the answers to several questions; default answers are fine for most users. Moreover, any settings can be changed afterward quite easily.

Once the initialization is complete, you will have a barebones repository layout:

```
Base Bundler Cfg Metadata Pkgmgr Rules SSHbase etc
```

Many of these directories start out empty; the main exception is the Metadata directory. It includes two files metadata.xml, which describes groups, and clients.xml, which includes client data. At this point you can start the Bcfg2 server. The server process will begin processing filesystem events, reporting either gamin or fam events depending on which file monitoring system is in use. All example logs quoted here were generated on a system using gamin for file monitoring.

```
> /usr/sbin/bcfg2-server
Bound to port 6789
Failed to read properties file; TGenshi properties disabled
Suppressing event for file python-mysql.xml~
Suppressing event for file x11-dev.xml~
Processed 109 gamin events in 0.422 seconds. 0 collapsed
Suppressing event for file converted.xml~
Processed 1861 gamin events in 19.829 seconds. 0 collapsed
Processed 121 gamin events in 0.240 seconds. 0 collapsed
Processed 26 gamin events in 0.148 seconds. 0 collapsed
```

The server logs to both standard out and syslog by default. The same messages are replicated in both locations. As we mentioned in the previous chapter, the Bcfg2 server uses file monitoring to maintain a coherent cache of file contents in memory for performance reasons. It maintains an internal set of file name patterns that are ignored; these typically correspond to editor temp files and the like. Whenever an event is ignored, the server will describe it.

When new events are available, bcfg2-server wakes up and processes them, rereading files if needed. Upon server startup, all files need to be read. On a system with a medium to large repository, startup can take 30 to 40 seconds; however, with an initial repository such as the one we just created, start up should be instantaneous. Once no events have been reported for a few seconds, the server becomes available to clients. Subsequent repository modifications result in additional file change events. For example, the command:

```
> touch /var/lib/bcfg2/Metadata/clients.xml
```

results in the server message:

```
Processed 1 gamin events in 0.118 seconds. 0 collapsed
```

At this point, the server is ready to serve configuration goals to clients.

3.2 The Client

The Bcfg2 client package is installed in the same way as the server package, only it usually has fewer dependencies. It is a good idea to install the client on the same machine as the server package, since you are eventually going to want to manage it using Bcfg2 as well.

The initialization process done above will set you up with one client machine: the machine on which the Bcfg2 server is running. The default distro group chosen during the initialization process is also set up as the default profile for new clients.

3.2.1 Bootstrapping New Clients

Bcfg2 clients are able to bootstrap themselves through command line arguments without using a configuration file. In the following example, the Bcfg2 client connects to the server that we have set up entirely using arguments:

```
> bcfg2 -x <password> -S https://<server>:<port> -F <fingerprint> -v -n
```

This command runs the client, manually specifying communication parameters: password, a server URL, and an SSL certificate fingerprint. The password and server URL were chosen as a part of the repository initialization. The fingerprint is used to validate that the client is talking to the expected server. It can be generated from the server certificate by running the following command on the server:

```
> bcfg2-admin fingerprint
e91f419beba103ae6bed7ad98673f4fddc59dda1
```

As we mentioned previously, the -F flag is not implemented in versions of Bcfg2 prior to 0.9.6; it can be omitted, but server fingerprint verification will not occur. In these versions, the server fingerprint can still be specified as a configuration file option.

Note that the use of the -x flag exposes the password to other users through the use of ps. Likewise, commands can be saved in shells' history files. Certainly, care in the use of this feature is a good idea.

In addition to the communication parameters, this invocation of the Bcfg2 client specifies two operational options: -v enables verbose mode, which displays a large amount of information about the operations of the client, and -n specifies dry-run mode. This is one of the most frequently used Bcfg2 client options. It runs the server through the entire configuration generation process, examines the configuration it gets back, and uploads a comparison of its state with the goals to the server. In verbose mode, it displays summary statistics as well. In dry-run mode, the Bcfg2 client performs exactly the same analysis steps as it would in regular mode, with the exception of making any changes to the client system. Because it updates the central reporting database with current state information, dry-run mode is useful for cases even where administrator supervision is desired; it detects when goals have diverged from current system state.

Once the client has been run, it will produce output that looks much like the following:

```
Loaded tool drivers:
APT       Action      DebInit    POSIX

Phase: final
Correct entries:          0
Incorrect entries:        0
Total managed entries:    0
Unmanaged entries:        2308
```

This display summarizes the current state of the client and the tool drivers available to the client. Correct entries are entries that conform to goals. At this point, the number of the goals ("Total managed entries") is 0, so 0 correct entries are expected. Likewise, no goals do not conform. The final number, unmanaged entries, are entries that exist on the client (and can be detected), but were not included in goals. Over time, we will reduce this number and increase the above three.

3.3 Making Changes with Bcfg2

Up to this point, the processes described have been entirely passive; no changes have been made to client configurations. That is all about to change.

3.3.1 Managing Configuration with Bcfg2

Managing configuration with Bcfg2 consists of specifying server-side configuration rules that describe configuration goals for clients. The formulation of these rules has three basic steps. These steps are the same, whether the changes will affect configuration files, packages, or services and regardless of the number of clients involved in the operation; rules that apply to one client are specified in the same way as rules that apply to one thousand clients.

- ❖ Target group isolation: When undertaking any configuration task, you must first determine which clients should be affected by this change. Is it all clients, or clients in a particular group? If the answer is no to both of these questions, you might need to define a new group and add it to the group hierarchy. The chosen group will be needed to formulate configuration rules.
- ❖ Entry addition: Next, you must determine which, if any, configuration entries will be added to client configurations. If additions are needed, they must be described in a configuration rule.
- ❖ Entry contents: Finally, in any configuration change, there will be one or more content rules changing the goal entry contents for the target client group. These rules must be specified.

3.3.2 Managing /etc/bcfg2.conf

Although the manual specification of all communication parameters from the command line is perfectly functional, it is inconvenient. As a first example of configuration management, we will begin to manage /etc/bcfg2.conf using Bcfg2.

Following the three steps described in the last section, first we determine which clients should be affected by this operation. We want all clients to receive an instance of /etc/bcfg2.conf that includes all of the communication parameters specified in the command line. So, these rules will apply to all clients.

Next, we need to determine if any entries will be added to the configuration goals because of these new rules. Because the goals are currently empty, the answer is clearly yes. We want to add the configuration file /etc/bcfg2.conf to all client goals. To accomplish this, create a new file called /var/lib/bcfg2/Base/basic.xml with the following contents:

```
<Base>
   <ConfigFile name='/etc/bcfg2.conf'/>
</Base>
```

Bcfg2 includes a repository validator. After any modification of XML files in the Bcfg2 repository, the following command should be run to ensure that the repository is well-formed:

```
> bcfg2-repo-validate -v
/var/lib/bcfg2/Metadata/clients.xml checks out
/var/lib/bcfg2/Base/basic.xml checks out
/var/lib/bcfg2/Metadata/groups.xml checks out
```

At this point, if the client is run again, the server will correctly add /etc/bcfg2.conf to the client goals but will be unable to bind contents to the goal. It will produce an error message:

```
Failed to bind entry: ConfigFile /etc/bcfg2.conf
Generated config for ubik in 0.00308299064636 seconds
```

Likewise, the client will detect that the configuration is not properly formed and will issue an error message as well:

```
Incomplete information for entry ConfigFile:/etc/bcfg2.conf; cannot verify
    ... due to absence of owner:group:perms attribute(s)

In dryrun mode: suppressing entry installation for:
    ConfigFile:/etc/bcfg2.conf

Phase: final
Correct entries:            0
Incorrect entries:          1
    ConfigFile:/etc/bcfg2.conf
Total managed entries:      1
Unmanaged entries:          2308

RecvStats completed successfully
```

Finally, we will add a rule that describes the contents of /etc/bcfg2.conf. The following commands create an area to manage the configuration file using the Cfg plugin, create a default version of the file, and set the file permissions to be owner-read-write only. Create a file /var/lib/bcfg2/Cfg/etc/bcfg2.conf/bcfg2.conf with the following contents:

```
[communication]
protocol = xmlrpc/ssl
password = <password>
fingerprint = <fingerprint>

[components]
bcfg2 = <server URL>
```

To control file installation metadata, create a second file called /var/lib/bcfg2/Cfg/etc/bcfg2.conf/:info with the following contents:

```
owner: root
group: root perms: 600
```

At this point, we can run the Bcfg2 client again, with the same options, still in dry-run mode, and it detects that there is an entry that should be updated. In the statistics message, one entry is now incorrect and one entry is managed. The Bcfg2 client includes an interactive mode that provides information about changes that will be performed. This can be activated by changing -n to -I. It steps through each change, providing detailed information, and allows the administrator to choose changes individually for installation. Interactive output will include the following text:

```
ConfigFile /etc/bcfg2.conf does not exist
Failed to read /etc/bcfg2.conf: No such file or directory
```

```
...
Install ConfigFile: /etc/bcfg2.conf? (y/N):
```

After answering y to the question, the Bcfg2 client will install /etc/bcfg2.conf. Once this is done, the -F, -x, and -S options can be removed from the command line when calling Bcfg2.

3.3.3 Further Refinement

Configuration goals change over time, usually due to either new information or a wider perspective. The rules specified in the last subsection manage the case described quite well; all clients get an identical version of /etc/bcfg2.conf. However, the clients that we already have don't correspond to that goal. The Bcfg2 server needs a more complicated version of /etc/bcfg2.conf, describing various server options such as repository location and which plugins should be enabled.

If the Bcfg2 client is run on the server host, it will attempt to replace /etc/bcfg2.conf. We can now demonstrate the ability of Bcfg2 to pull configuration content rules from clients, in this case the Bcfg2 server itself. If we run the Bcfg2 client on the server host in dry-run mode, it will upload statistics to the server describing the current state of the client. These statistics contain detailed information about all incorrect entries, in this case /etc/bcfg2.conf. After running the client, the server has all information to refine configuration rules appropriately.

```
> bcfg2-admin pull ubik ConfigFile /etc/bcfg2.conf
Found 4 entries for ubik:ConfigFile:/etc/bcfg2.conf
Found entry from Thu Feb 28 20:01:57 2008 Located diff:
...
Should this change apply to all hosts affected by file
      /var/lib/bcfg2/Cfg/etc/bcfg2.conf/bcfg2.conf? (N/y): n
This file will be installed as file
      /var/lib/bcfg2/Cfg/etc/bcfg2.conf/bcfg2.conf.H_ubik
Should it be installed? (N/y): y
writing file, /var/lib/bcfg2/Cfg/etc/bcfg2.conf/bcfg2.conf.H_ubik
```

This step has added a new rule to the repository governing the contents of /etc/bcfg2.conf. This rule applies only to the host "ubik" and overrides the default rule. At this point, all clients, including the Bcfg2 server host, will report one managed and correct entry.

3.4 The Next Steps

This chapter has given a general overview of using the Bcfg2 tools from beginning to end. In the next couple of chapters we will take a closer look at specific tasks and how to implement them using Bcfg2 semantics. These tasks start out simple, making them perfect starting points to tackle during a few free minutes during the day. They then quickly progress to more complex tasks that can be used as a basis for powering your entire organization.

4. Task Examples

One of the most difficult aspects of using a new tool productively is getting past the initial learning curve; this chapter is specifically aimed at easing that problem. Each of the following sections describes a task that will help you use Bcfg2 to bring a network of machines under configuration management control. We will describe the thought process and considerations needed to complete each task, in a way that's specific enough to act as a recipe for getting the task done.

This chapter is meant to be both a learning tool and a reference tool. When first learning Bcfg2, the chapter serves as "Learn Bcfg2 in an Hour a Day," with each task acting as a single lesson that builds on the previous one. At the same time, each example is chosen for its flexibility: the skills learned from one task are immediately applicable to similar ones, and so are able to serve as both inspiration and guide when trying something new. For example, Section 4.1 can apply to any independent file that needs management, while Section 4.4 is a great general-purpose starting-point for almost any nontrivial task.

During our discussions of tasks, we will refer to the example clients.xml and groups.xml files shown in Figures 4.1 (below) and 4.2 (next page), respectively.

```
<Clients version="3.0">
    <Client name="www.example.com" profile="web-server" />
    <Client name="mail.example.com" profile="mail-server" />
    <Client name="gromit.example.com" profile="desktop" />
    <Client name="max.example.com" profile="desktop" />
    <Client name="victor.example.com" profile="desktop" />
</Clients>
```

Figure 4.1: A simple clients.xml

4.1 Message of the Day

Our first task is simple: to manage a uniform /etc/motd file across all of our machines. This file is often used to state site policy, warn users of upcoming outages, or remind users of useful commands that they may need. Managing its contents in a central location prevents mistakes or omissions when deploying this file across the network and makes updates easy. It is also a relatively harmless file to modify, and is therefore a great place to start exploring configuration management. We'll make use of the decision steps in Section 3.3.1 while planning and making the changes.

First, identifying which clients will be affected by this change should be easy: we want all of our machines to have a managed message of the day. Note that the message doesn't necessarily need to be the *same* for all of the machines; however, we do want to manage the file on all of them.

```
<Groups version="3.0">
    <Group name="web-server" profile="true">
            <Group name="server"/>
            <Bundle name="apache"/>

    </Group>
    <Group name="mail-server" profile="true">
            <Group name="server"/>
            <Bundle name="postfix"/>
    </Group>
    <Group name="desktop" profile="true">
            <Group name="basic"/>
            <Bundle name="X11"/>
            <Bundle name="gnome"/>
    </Group>
    <Group name="server">
            <Bundle name="tripwire"/>
    </Group>
    <Group name="basic">
            <Bundle name="ssh"/>
            <Bundle name="ntp"/>
            <Bundle name="motd"/>
    </Group>
</Groups>
```

Figure 4.2: A simple groups.xml

Next, we need to add the motd configuration element to the server's repository of known configuration elements. We'll do this by creating a very simple motd Bundle, which can be seen in Figure 4.3.

```
<Bundle name="motd">
    <ConfigFile name="/etc/motd"/>
</Bundle>
```

Figure 4.3: A message-of-the-day Bundle, from Bundler/motd.xml

This Bundle was already included in the base group in Figure 4.2, so by just dropping the Bundle into Bundler/motd.xml and the file itself in Cfg/etc/motd/motd, we've begun managing a uniform system-wide message of the day file. The client will put this file in place on each client machine the next time it runs.

Finally, let's consider the contents of the entry. Since /etc/motd is a static configuration file, we'll make use of the Cfg plugin to install it. By default the Cfg plugin stores its repository of files in /var/lib/bcfg2/Cfg/. While this directory starts out empty in a new server installation, a populated Cfg repository mimics the root (/) directory of our managed machines. Thus, our standard motd file will be kept in Cfg/etc/motd/motd. Note that the file is actually stored in a directory of its same name. We will see in future tasks how to use this structure to store group- and machine-specific configuration files, as well as other data.

Once this is set up, the Bcfg2 client can be used to install the new /etc/motd file. The -I flag prompts for each entry to be modified, adding details about the change:

```
# /usr/sbin/bcfg2 -I
Failed to read /etc/motd: No such file or directory
Install ConfigFile: /etc/motd? (y/N): y
```

4.2 An NTP Client

A slightly more complex task is to manage a network time protocol client. In this case we have three configuration elements to worry about: a package (ntp), a configuration file (/etc/ntp.conf), and a service (ntpd). The Bundle for these three items is shown in Figure 4.4. By placing these three elements together in a Bundle, we gain two notable benefits over treating them as independent items. First, we have created a simple building block that can be applied to almost any system. There's no need to list all three items for any new classes of machines; instead the new classes just need to include the NTP Bundle. Secondly, this gives the Bcfg2 client a hint to treat these three elements as a group. In this case, when it installs a new version of the ntp package or an updated version of ntp.conf it will deduce that it should also restart the ntpd service so the changes take effect.

```
<Bundle name="ntp">
     <Package name="ntp"/>
     <ConfigFile name="/etc/ntp.conf"/>
     <Service name="ntpd"/>
</Bundle>
```

Figure 4.4: An NTP Bundle

Since we have a service included in this Bundle, we have to take one extra step—specifying which types of machines we want the service to actually run on. In the Svcmgr directory in the Bcfg2 repository, we'll create a services.xml file as seen in Figure 4.5.

```
<Services priority="0">
     <Group name="base">
               <Service name="ntpd" type="chkconfig" status="on"/>
     </Group>
</Services>
```

Figure 4.5: A simple services.xml

Since we want to run NTP on all of our machines and since we included the base group in all other groups in Figure 4.2, placing the service definition in the base group causes all clients to turn on the NTP service.

Assuming you have a working Pkgmgr setup, the final step needed to fully manage NTP clients is to place your ntp.conf in Cfg/etc/ntp.conf/ntp.conf. The next time each client runs they will install the NTP package (if needed), put the managed ntp.conf in place, and turn on the ntpd service. If you switch to a different NTP server in the future, updating all of your machines is as simple as updating the ntp.conf file in the Bcfg2 repository and running the Bcfg2 client on all machines (which can be done nightly via cron). When the new file is installed the service will be restarted, taking the guesswork out of which machines are updated and which ones might still be pointing at an outdated server.

4.3 Managing the Base Configuration

Sections 4.1 and 4.2 showed examples of Bcfg2's basic building block, the Bundle. Bundles are used to collect similar or interdependent configuration elements together to make managing and reusing those elements painless. However, on modern operating systems there is often a very long list of packages that don't need to be wrapped together in Bundles, usually because they are part of the base operating system install and are definitely going to exist on all machines. On a current SLES 10 system, for example, this includes such packages as aaa_base, bash, and rpm. We use Bcfg2's Base to manage these sorts of independent configuration elements.

Figure 4.6 shows a *very* simple Base file containing these packages and a couple of extra entries. Its contents look similar to the Bundles in Figures 4.3 and 4.4, but with different surrounding tags. As mentioned in Section 4.2, grouping configuration elements together in a Bundle gives the client hints that it should check for interdependencies between all of the elements in the Bundle. Base is for all of those independent elements that aren't really related to other elements. The elements listed in Figure 4.6 are all such elements: they're needed on every system, and when one of them changes there's no need to recheck any of the others.

```
<Base>
    <Package name="aaa_base" />
    <Package name="bash" />
    <Package name="rpm" />
    <Directory name="/sandbox" />
    <ConfigFile name="/etc/raidtab" />

</Base>
```

Figure 4.6: A simple Base file

So what goes in Base? When you are setting up Bcfg2 for the first time, generating a complete list of every package on your first system is a great place to start. By putting all of these Package entries in a Base file, you've made a *huge* jump toward managing your system—on typical current machines, this can be hundreds of packages. From that starting point, it is then much easier to compare this specific-to-one-machine Base to others, paring down packages that don't match up between all of the machines in your network. These can then be left in Base and installed on all machines, leading to consistency; pulled out into Bundles along with their dependencies; or put into group-specific sections in Base. Whatever their final location, starting with a complete package list in Base is a useful first step in the incremental configuration-building process.

4.4 An NTP Client and Server

In Section 4.2 we created a complete Bundle for managing NTP clients, but we can easily extend this work to manage an NTP server, too. The only difference between an NTP client and an NTP server is its configuration file. Both use the same ntp package, run the same ntpd daemon, and read from an ntp.conf configuration file, but the NTP server needs a different version of that configuration file from that of all of the clients. We can easily set up a server-specific NTP configuration file by creating a new ntp-server group, associating our NTP server machine with that group, and then placing a group-specific file in the Cfg repository.

The first thing to do is to create an NTP server ntp.conf configuration file. To create group-specific config files for Bcfg2, you append .GNN_groupname to the filename, where NN is a two-digit numeric priority and groupname is the group to which this file is specific.

```
Cfg/etc/ntp.conf/
    ntp.conf
    ntp.conf.G90_ntp-server
```

Figure 4.7: A group-specific ntp.conf file

Figure 4.7 shows the contents of our Cfg/etc/ntp.conf/ directory after this file is created. The priority of 90 was chosen rather arbitrarily: if a client matches multiple group-specific files in the directory, then the one with the highest priority number is handed down to the client. In this example, if there was also an ntp.conf.G80_mail-server file in the directory, then our new priority 90 file would trump that one for the client mail.example.com.

Looking back at our clients.xml file in Figure 4.1, we only have two machines on which we would want to run our local NTP server: either www.example.com or mail.example.com. Let's choose to run it on mail.example.com.

```
<Group name="mail-server" profile="true">
    <Group name="server"/>
    <Group name="ntp-server"/>
    <Bundle name="postfix"/>
</Group>
```

Figure 4.8: An update to groups.xml

Figure 4.8 shows an updated mail-server group section from Figure 4.2. Since the mail-server group already gets the ntp Bundle (through the server group, which in turn gets it through the base group), all we need to do is add the ntp-server group to the mix.

With the new file in place and the update made to groups.xml, the next time mail.example.com runs the Bcfg2 client it will get its group-specific configuration file, while all other clients will be given the default, non-specific file. Any number of group-specific files can be created in this same way, but recall that each client will always only receive the most specific, highest-priority file it matches.

4.5 Managing an SSH Infrastructure

SSH host keys are an obvious set of files that can benefit from being managed by a centralized system. This is especially apparent to anybody who has been presented with the familiar remote host identification has changed message after rebuilding a machine and using SSH to log in to it again. This happens because the SSH server generates a new key on the newly rebuilt machine, causing your SSH client to doubt the authenticity of the machine to which it is connecting. The easy way to fix this problem is to import each key into Bcfg2 as a host-specific file. Then when a machine is rebuilt, running Bcfg2 on it the first time will put the old key right back where it belongs. But it turns out we can do even better than that.

Bcfg2 ships with a plugin named SSHbase, named after its role: an SSH key database. It extends Bcfg2's normal abilities to handle all aspects of SSH key management:

- ❖ If a machine's keys are already stored in SSHbase, they are handed out in the configuration as the canonical versions.
- ❖ If a machine's keys are not in SSHbase, it is assumed that the machine is a new build. New keys are generated, automatically placed in the database, and treated as canonical for that machine from then on.
- ❖ If desired, a complete ssh_known_hosts file can be generated for each machine, containing the public keys of all hosts SSHbase knows about. This greatly simplifies setting up SSH host-based authentication.

To begin using the SSHbase plugin, we first need an SSH Bundle. An example is shown in Figure 4.9.
As with any Bundle, the example contains all of the pieces that are needed for an SSH

```
<Bundle name="ssh">
    <ConfigFile name="/etc/ssh/sshd_config"/>
    <ConfigFile name="/etc/ssh/ssh_config"/>
    <Package name="openssh"/>
    <Service name="sshd"/>
    <!-- The below are handled by SSHbase -->
    <ConfigFile name="/etc/ssh/ssh_host_dsa_key"/>
    <ConfigFile name="/etc/ssh/ssh_host_dsa_key.pub"/>
    <ConfigFile name="/etc/ssh/ssh_host_rsa_key"/>
    <ConfigFile name="/etc/ssh/ssh_host_rsa_key.pub"/>
    <ConfigFile name="/etc/ssh/ssh_host_key"/>
    <ConfigFile name="/etc/ssh/ssh_host_key.pub"/>
    <ConfigFile name="/etc/ssh/ssh_known_hosts"/>
</Bundle>
```

Figure 4.9: An SSH Bundle

service to act correctly: host keys, configuration files, SSH service, and SSH package. By default, all ConfigFile entries are handled by the Cfg plugin, in which case we would need to place all of the ConfigFile entries mentioned in the SSH Bundle in their proper places in the Cfg/ directory in the Bcfg2 repository. Instead, however, we will use the SSHbase plugin to manage these files. To turn on this plugin, we simply add SSHbase to the generators list in our server's bcfg2.conf file. Make sure to make this change in the version of this file in the Cfg repository, or this change might be overwritten.

With the SSHbase generator enabled, the Bcfg2 server treats the ConfigFile entries marked in Figure 4.9 differently. When a client contacts the Bcfg2 server for the first time, the server checks whether or not a complete set of public and private keys exists for it. If not, it generates a set, saves them to the repository, and includes them in the initial configuration passed back to the client. This is especially useful for dynamic sets of machines, such as desktop workstations, which then have their unique SSH keys managed from their very first boot without any manual intervention. Meanwhile, the ssh_known_hosts file is always kept up-to-date with the latest set of keys for all known machines. This keeps anything that relies on that file, such as host-based authentication, working smoothly even in a highly dynamic environment.

When deploying SSHbase, acquisition of pre-existing keys is a good idea. bcfg2-admin pull can be used to install these keys into the SSHbase repository, based on client statistics uploads:

```
for file in ssh_host_dsa_key ssh_host_rsa_key ; do
    for suffix in “” .pub ; do
            bcfg2-admin pull -f \<client\> /etc/ssh/${file}${suffix}
    done
done
```

4.6 Using Actions

Some configuration changes require actions to take place on a client after the related files or packages are installed. For example, after a new aliases file is installed for the Postfix mail server, it is necessary to run the newaliases (or similar) command to generate the new aliases database. Bcfg2 supports this need through the use of Actions.

Figure 4.10 shows a Postfix Bundle that uses an Action to perform the needed local change.
This Bundle contains many of the elements we have already seen: a Package entry, a

```
<Bundle name=”postfix” version=’2.0’>
    <Package name=”postfix”/>
    <Service name=”postfix”/>
    <ConfigFile name=”/etc/postfix/main.cf”/>
    <Group name=”mail-server”>
            <ConfigFile name=”/etc/postfix/virtual”/>
            <Action name=”postmap”/> <
    /Group>
</Bundle>
```

Figure 4.10: A Postfix Bundle using Actions

Service entry, a ConfigFile entry, and a group-specific section. It also contains an Action tag, which only contains an Action name. The Action itself is defined in a separate file, located in Rules/actions.xml in the Bcfg2 repository’s top-level directory.

In Figure 4.11 we see what the actual Action definition looks like.

```
<Rules priority=”0”>
    <Action name=”postmap” command=”/usr/sbin/newaliases” \
            when=”modified” timing=”post” status=”ignore”/>
</Rules>
```

Figure 4.11: An Actions file

In this case our action is named postmap, its name in the Postfix Bundle, and the command it runs is /usr/sbin/newaliases. The next three options fine-tune the Action for the specific need at hand. Here we're running the action whenever a Bundle element is modified (when="modified"), having the Bundle run after all its elements are installed (timing="post"), and not reporting the exit status of the action to the reporting system (status="ignore").

With these two pieces in place—the Action tag in the Bundle and its definition in the Rules directory—our task is done. In this case, when a client runs it will examine all of the member elements of the Bundle and install any updates that are needed. If something changes, it will run the Action after all update configuration elements have been installed. Note that the Action will run regardless of what element has changed. In this case, that's fine, but in some cases we might want to do sanity checks within the Action command to make sure we really want it to run.

5. Troubleshooting

Any sufficiently sophisticated tool occasionally behaves unexpectedly; Bcfg2 is no different. For such occurrences, we have built several mechanisms to make Bcfg2's operations transparent. At a basic level, the server process logs to syslog; we will provide an overview of the data included. `bcfg2-info`, a diagnostic framework around the `bcfg2-server` code, has the ability to inspect the state of the server and its rules to produce partial or complete configuration goals without involving client systems. Finally, we describe the information made available by the Bcfg2 client. Each of these topics is discussed in turn, highlighting their major uses and capabilities. As one implements sophisticated configuration patterns, these capabilities quickly become indispensable. We will highlight their use in Chapter 6.

5.1 Logging

During ordinary operations, the bcfg2-server process produces log messages corresponding to a variety of normal events:

- Binding to the server port: After the server process has successfully bound to the IP port, it produces a message saying so.
- Processing file events: The server is notified whenever a file in the repository is modified. After processing these events, it produces a message telling how many events were processed. It also reports how many events were coalesced. Event coalescing occurs when the same event occurs repeatedly. For example, when a large file is written, several file-changed events may be created during the same write. These events will be coalesced and only processed once.
- Generating client configuration goals: When a client requests its configuration goals, the server reports this event, including the client's name and the elapsed time, upon completion.
- Client state reports: When the server receives statistics upload from clients, it reports the client's state (either clean or dirty) in a log message.
- Ignoring files matching the backup file pattern: The Bcfg2 server ignores files that end in ~ or .swp. Whenever it encounters and ignores an event for these files, it reports this occurrence.

Each of these indicates expected activity. The server also produces error messages in the case of runtime errors:

- Client metadata resolution problems: When a client connects, the server performs a resolution process to determine its metadata. This can fail in several cases. If a client is not already registered and no default profile is configured, the server cannot construct metadata for it. In this case, an error is reported and the client will fail, since it cannot continue.

- Entry content binding failures: When the server constructs client goals, it attempts to bind contents to each entry. This can occur in several cases. If no content rules are present, it will fail outright. In other cases, multiple content rules might be applicable, or multiple plugins might contain rules. If the Bcfg2 server recognizes an ambiguous situation, it will refuse to bind entry contents and will produce an error message. These errors can be associated with a client by finding the following configuration generation message.
- Communication setup errors: When the server starts up, it will produce error messages if it cannot successfully bind to the address requested. Similarly, if the certificate does not exist or is not readable, the Bcfg2 server will produce an error message.
- Client communication errors: Network transactions can fail in a variety of ways ranging from socket errors to data corruption. Because Bcfg2 uses HTTPS for client/server communications, transmissions are quite robust. If the server detects any problems of this sort, it closes the socket and reports an error.
- Filesystem I/O errors: In some cases, the server will get a filesystem event for a file that has been deleted between the event creation and its processing. In this case, the Bcfg2 server will fail to read the file. These errors can usually be ignored, but can be indicative of strange activities on the server.
- XML parsing failures: Several files in the Bcfg2 repo are XML files. If written incorrectly, they may fail to parse. When this occurs, the server reports the parse failure. `bcfg2-repo-validate` can be used to find both XML syntax errors and schema validation errors.

5.1.1 Common Errors

During normal operations, a variety of common errors can occur. This section explains several of them.

- `Suppressing event for file ssh.xml`: This error message is generated when the name of a file in the repository is recognized as a temporary or backup file. These messages are typically harmless.
- `Could not process filename /path/to/file/.#template.txt; ignoring`: This error message means that the server was unable to process the name of a file in the repository. In many cases, the roles of individual files are determined by their names. This issue can occur for a variety of reasons. Editors often use files with strange names as temporary storage; in this case, the files are often gone before the `bcfg2-server` process notices them. The misnaming of files can also cause a similar behavior. For example, when using Cfg, group-specific files must encode a priority so that ties can be broken for clients that are members of multiple groups. In this case, a file name of the form file.G_group will cause a similar error message to be displayed.
- `Failed to bind entry: ConfigFile /path/to/file`: This error message occurs when the configuration for a client includes an entry that is undefined for that cli-

ent. For example, a Bundle might include a ConfigFile entry that has not been defined in Cfg. Likewise, it might be defined, albeit not for the current client. When this error occurs, ensure that the entry is defined for the current client. This can be done by using the mappings, build, and buildfile commands provided by bcfg2-info, described in the next section.

- Templating failures: Templates are effectively user-defined programs for generating structure and configuration file contents. As with any program, bugs can occur in templates. When templates are buggy, the templating process will fail, resulting in a message like:

  ```
  NotFound: cannot find 'variable'
  TCheetah template error for /etc/diskinfo
  Failed to bind entry: ConfigFile /etc/diskinfo
  ```

  The first line describes the templating system error, the second describes the context, and the third is the general entry binding error message, described in the previous section.

5.2 bcfg2-info

bcfg2-info provides a diagnostic interface for the bcfg2-server logic. When it starts, it produces a series of messages that are similar to those produced by bcfg2-server; the only difference is that all communication messages are gone. Once bcfg2-info finishes processing all pending filesystem events, it leaves you at a command prompt. A dozen commands are available. Four describe client metadata. Four commands illuminate the configuration goal construction process. Two commands display the internal structure of configuration rules. Finally, two commands control the execution of bcfg2-info itself. These commands are far more useful than reading the repository, because they demonstrate the results of the bcfg2-server code processing the repository.

- clients: The clients command displays clients and their profiles.
- groups: The groups command builds a table of all groups, with a recursively built list of included groups. Each group is annotated with its profile and category status.
- bundles: The bundles command displays the list of Bundles included with each group.
- showclient: The showclient command displays client metadata for a series of clients, including profile and group information.
- build: The build command builds a complete set of client goals for a given client and places it in a file. These files can be used with bcfg2-admin compare to find goal differences.
- buildall: The buildall command builds individual client goals for each configured client and places each in the specified directory. This command is frequently used to test server code upgrades.
- buildfile: The buildfile command constructs a single ConfigFile entry for a client. This command is useful for performing point-wise inspections of the impact of new configuration rules. We frequently use this command for debugging new configuration file templates.

- showentries: The showentries command produces a list of goal entries for a given client.
- generators: The generators command produces a list of plugins that can provide content rules.
- mappings: The mappings command describes which plugins provide content rules for each goal entry.
- update: The update command causes bcfg2-info to process all pending filesystem events.
- debug: The debug command drops the user into a Python interpreter with all of the bcfg2-server data structures and code loaded. This is typically used by developers for low-level debugging.

The main use of bcfg2-info is understanding how configuration rules have been processed by the repository handling code.

5.3 Client Debugging

The Bcfg2 client logs to STDOUT and ERR. The verbosity level can be controlled through use of the -v and -d flags; these respectively increase and decrease the verbosity. When running in debug mode, users can expect to see the steps executed by the client as it is running, with extra information along the way. Information displayed includes:

- XML-RPC calls: Each XML-RPC call executed by the server on the client's behalf is displayed, with error information in event of a failure.
- Entry verification results: As the client verifies its current state, it individually verifies each goal. Detailed information about each failing goal is displayed.
- Installation steps: As goals are installed, detailed command information, output, and return codes are displayed.
- Statistics information: At the beginning of client execution, each step between goal installation, and the completion of goal installation, the Bcfg2 client details the quantities of goals in good and bad states and the number of goals that were modified during execution. This information is a subset of the information stored in the reporting system.

5.3.1 Common Client Errors

The client can emit a number of error messages during the course of normal operations. These include:

- no server x509 fingerprint; no server verification performed!: This error message occurs when the client has not specified a fingerprint for the server SSL certificate. This error can be remedied either through the use of the -F command line options or by specifying the server fingerprint in /etc/bcfg2.conf.
- The following entries are not handled by any tool:: This error occurs when the client detects an entry that no active tool driver handles. This problem can occur in two cases. Either the entry includes incorrect tag or type information, which leads the client not to be able to properly recognize it, or the appropriate driver has not been loaded. If the entry is incorrectly described, it can be

fixed in the server repository. If the appropriate driver has not been loaded, two causes are most likely. If a list of drivers is being explicitly listed (using -D or the equivalent in bcfg2.conf), then the necessary driver may not be enabled. If the needed driver is explicitly listed, then it is possible that prerequisite tools are not available on the system. For example, the APT driver requires the dpkg, apt-get, and debsums commands.

- Incomplete entries: This error occurs when an entry in the client goals does not include sufficient information to verify the entry. In this case, it will be neither verified nor installed, and will be assumed to be incorrect. This error usually corresponds to the Failed to bind entry server-side error.
- Incomplete information for entry ConfigFile:/etc/foo.conf; cannot verify ... due to absence of owner:group:perms attribute(s): This error can occur in two ways. Different amounts of information are required for verification and installation; it is possible that an entry includes enough information for verification, but not enough for installation. Each message describes the missing attributes.

6. Advanced Task Examples

The tasks in Chapter 4 show how to import a number of everyday configuration elements into Bcfg2's realm of control, and these tasks cover the concepts needed for about 90% of configuration elements. However, there are always more complex elements that require configuration management. In this chapter we will explore some more complex tasks involving dynamic data, probing clients, and similar concepts.

6.1 Templates: Setting a Hostname

The tasks in Chapter 4 all relied on static files. While the majority of configuration files are static, at times it is a great advantage to be able to generate files on the fly. Bcfg2 supports two templating engines for these purposes, Cheetah and Genshi, each of which has its own merits. These two templating engines are their own full projects and their documentation is outside the scope of this booklet, but each comes with a full set of documentation and examples with which to get started.

Before using either template engine you will need to download and install them from their respective project pages. After installation, adding `TCheetah` or `TGenshi` to the `generators` line in your bcfg2.conf will enable the template engine plugins during the next server restart.

Most GNU/Linux distributions use a file in /etc/ to store the machine's hostname; SLES, for example, uses /etc/HOSTNAME. This file is usually set at build time and can be updated by DHCP clients or other scripts to make sure it always makes sense. In this case, we want to make sure it is always the same name that the Bcfg2 server knows the machine by. Since the contents of this file would be different on every machine we manage, keeping a giant directory full of host-specific static files in the Cfg/ directory would quickly expand beyond control. By using templates, however, we can simply grab the hostname that the Bcfg2 server already knows and use that as the contents of the file.

For this task we'll use the Cheetah templating engine, as it is built more for flat text files than Genshi. We'll start by writing the simple template itself, the complete listing of which is seen in Figure 6.1.

Contents of TCheetah/etc/HOSTNAME/template:
```
$self.metadata.hostname
```

Figure 6.1: The HOSTNAME template

Clearly, this is a very simple template. It consists of only one line, a variable that it inherits from the Bcfg2 server itself (see Table 6.1 for a complete list of built-in variables that are available). In practice any Cheetah constructs can be used in a template, including blocks of Python code, but in this case we only need that one line.

Name	Type	Description
hostname	String	Host name
bundles	List	Names of Bundles included in this machine's configuration
groups	List	Names of groups to which this machine belongs
categories	List	Names of categories to which this machine belongs
probes	Dictionary	Dictionary of probe data keyed by probe name
uuid	String	This machine's UUID
password	String	The password this machine provided to the server

Table 6.1: Metadata available to templates

When working with static files and the Cfg plugin, we put the configuration files in a logical place in Bcfg2's Cfg/ directory. Templates follow a similar but slightly different layout: in this case, the file will be named TCheetah/etc/HOSTNAME/template. Note that the named directory structure is preserved, but with templates the file itself is named template. This reflects the fact that this file is special and not an exact, static file as with Cfg. Another notable difference from Cfg is that templates don't have group- or host-specific files—all of this logic is up to the template writer using Cheetah constructs and the server-provided variables self.metadata.hostname and self.metadata.groups. For example, if we wanted all machines in the web-server group to think their hostname was www.example.com (a very silly thing to do), we could use the template shown in Figure 6.2.

```
#if 'web-server' in $self.metadata.groups
www.example.com
#else
$self.metadata.hostname
#end if
```

Figure 6.2: A rather silly HOSTNAME template

The final piece we need is exactly the same as using static files with Cfg: we need to add a ConfigFile entry to an appropriate Bundle. For completeness this entry is shown in Figure 6.3, but it is left as an exercise for the reader to decide on a good Bundle to put it in.

```
<ConfigFile name='/etc/HOSTNAME'/>
```

Figure 6.3: A HOSTNAME ConfigFile entry

With this template in place, a fresh copy of the file's contents will be generated each time a client asks the server for its configuration. Since the contents are generated serverside, to a client it will appear as just another static file in its configuration.

6.2 Probes

Templates are a great way to put server-side dynamic data into configuration files, but sometimes you need to know something about the client before you can generate that data. Bcfg2 uses the concept of Probes for this purpose. A Probe is a shell script that is sent to the client and run there before the client's configuration is generated. The results of Probes are then passed back up to the server, where they can be accessed by templates as a part of the client's metadata.

Warning: This example has the potential to make your system unbootable if used incorrectly. During testing you should replace all instances of /etc/fstab with a safe file (e.g., /tmp/fstab) and ensure that its contents are 100% correct before using it on a live system.

Imagine you have to manage a set of machines that were purchased at two different times. Half of them have a second disk in them that is just used as scratch space and mounted on /scratch, while the others only have one system disk in them. These systems could be mixed all about in the infrastructure, so simple grouping won't help to keep track of which is which. How would you keep a sane /etc/fstab on all of them?

Figure 6.4 shows the contents of a script that counts the number of IDE devices on a system, printing out that number on STDOUT. Depending on the hardware involved, this can work perfectly for differentiating the two classes of machines: those with two IDE devices (one hard drive and one CD-ROM drive) don't need a scratch entry in /etc/fstab; those with three do. This file lives in the Bcfg2 repository in Probes/countdisks. The file name is the name by which the Probe's results will be called in a template.

```
#!/bin/sh
ls /proc/ide/ | grep hd | wc -l
```

Figure 6.4: A disk-counting Probe

Figure 6.5 shows the contents of such a template, which should make sense after our work in the previous chapter. Two things to note: first, the self.metadata.probes variable is a dictionary that holds results from all Probes that ran and is keyed on the name of the Probe. Second, the probe data is stored as a string, so we explicitly have to turn it into an integer with Python's int() function. This file, of course, goes into the Bcfg2 repository in TCheetah/etc/fstab/template, replacing any copy we might have already had in the Cfg/ directory. Don't forget to also add /etc/fstab to a Bundle or Base; if you don't do that, Bcfg2 won't bother handing it out to any clients.

```
proc        /proc           proc      defaults          0   0
/dev/hda1   /               ext3      defaults          0   1
/dev/hda5   none            swap      sw                0   0
/dev/hdc    /media/cdrom0   iso9660   ro,user,noauto    0   0
#if int($self.metadata.probes["countdisks"]) > 2
/dev/hdb1   /scratch        ext3      defaults          0   1
#end if
```

Figure 6.5: A template making use of Probe data

Once these pieces are in place, the chain of events that happens when a client runs is:

1. The client runs, contacting the server for data.
2. The server replies, handing down the list of Probes that need to run.
3. The client runs the Probes, returning the output of each to the server.
4. The server generates the client's configuration, running each involved template in turn.
5. When the /etc/fstab template runs, it checks the output of the countdisks Probe and includes the extra fstab line as needed.
6. The server passes the full configuration down to the client.
7. The client installs its configuration as it normally would.

And with that we have a fully automated backend in place that can dynamically add information to a client's fstab as needed.

6.3 Structure Templates

The first two sections in this chapter examine how templates dynamically generate configuration files for clients, but Bcfg2 has the ability to use templates for dynamic server-side structures too, creating, for example, dynamic Bundles that are updated by outside information. In this task we will look at a Bundle that creates home directories for users on machines that don't mount home directories from a central homes server. This can be useful on stand-alone management machines that must be accessible even if network homes aren't.

Bcfg2 uses the Genshi templating library and its strong XML connections for its Structure templates. Just as Cheetah templates go in Bcfg2's TCheetah/ directory, Structure templates go in SGenshi. However, since they are not attached to a configuration file path, they all live in that single top-level directory. In that respect, TCheetah/ can be looked at as the templating analog to Cfg/, while SGenshi/ can be seen as the analog to Bundler/. Don't forget that to be able to use SGenshi you'll have to install the Genshi templating engine and enable it by adding SGenshi to the structures line in /etc/bcfg2.conf.

Accounts management is outside the scope of this book, so we will make the assumption that our organization already has a robust accounts management interface in place and that it exports an excerpt from the password file containing just administrator accounts to the etc/passwd.users.management file in the Bcfg2 repository. This task can easily apply to different file locations and formats with minimal changes.

So, what do we need to make this work? If the list of users were static, it would be easy: we would just create a Bundle containing a Directory tag for each user. In this case we'll do the exact same thing, only using an SGenshi template.

Figure 6.6 (next page) shows the contents of one such template. The top-level tag declares it to be a Bundle, but note the addition of the XML namespace option. This option doesn't affect the way Bcfg2 itself handles the file, but it does mark it in a way that the Genshi engine will understand. The largest part of the actual content of the file is a block of raw Python code that reads the file, parsing out user names and storing them in a list. At the bottom is a Genshi for loop that generates the actual Directory entries for each user in the list. Note the if statement that surrounds the loop—we only want to do these actions on machines that are in the management group. Saving this file as SGenshi/adminhomes.xml in the Bcfg2 repository completes this task. With that in place we can treat the entire contents as a Bundle, adding it to groups in clients in groups.xml just like any other Bundle.

```
<Bundle name='adminhomes' xmlns:py="http://genshi.edgewall.org/">
    <?python
      if 'management' in metadata.groups:
        ismgt=True
        users = []
        passwd = open("/var/lib/bcfg2/etc/passwd.users.management")
          .readlines()
        for line in passwd:
          if(line.startswith('#')):
            continue
          splitline = line.split(":")
          users.append(splitline[0])
    ?>
    <py:if test="ismgt">
      <Directory py:for="user in users" name='/home/${user}' owner='${user}'
        group='users' perms='0755'/>
    </py:if>
</Bundle>
```

Figure 6.6: An SGenshi template

6.4 Complex Templates

DHCP is the standard protocol used to dynamically configure a machine's network addresses, and it works well from both a usage standpoint and a configuration management standpoint: no local state means that all clients can have the same static network configuration files. However, some machines are too important to rely on the DHCP service to always work (who hands out a DHCP address to the DHCP server? What if the DHCP server can't be rebooted because the secure admin gateway couldn't get an address last time it booted?), and their static configurations can become cumbersome to maintain. The obvious answer to this problem, of course, is to generate their static network configurations on the server using templates.

Some GNU/Linux distributions, such as SLES, throw a wrench into the nice simple world of templates, though: in SLES 10, for example, the most stable way to store static network configurations is in files that contain the MAC address of the corresponding interface in their file names. These files are all stored in /etc/sysconfig/network/ on each machine, as shown in Figure 6.7.

```
> ls /etc/sysconfig/network/
ifcfg-eth-id-00:10:18:2b:53:71   ifcfg-eth-id-00:14:5e:5a:a6:19
ifcfg-eth-id-00:10:18:30:96:1f   ifcfg-eth-id-00:14:5e:5a:a6:1b
```

Figure 6.7: Some SLES network configuration file names (applies to other distributions as well)

Since different machines will have different file names, it is clear that a simple template isn't going to do the entire job. We're going to need a Structure template, too.

But what about the file names? They're unique on each machine, so the Structure template needs to know the MAC addresses of the interfaces on each client to be able to generate the correct file names. We know how to do this: using a Probe, we can query the machine for a list of all of its MAC addresses before the Structure template gets parsed.

So now we just have one more problem: when the client runs, the server will pass it a Probe to run. This Probe will return a complete list of MAC addresses on the machine, and the Structure template will use this information to generate a dynamic list of ConfigFile entries which will then be handled by the configuration file template engine to insert the correct IP address information. The problem is that when the ConfigFile entries are created, the Bcfg2 server will start looking for a file to hand back that matches the entire file name, including the MAC address. It would seem that we are back where we started, needing to create a large repository of ConfigFiles, each with a unique name. Fortunately, Bcfg2 has a way to remap configuration element names for the purpose of data binding: the altsrc tag option.

When used with ConfigFile entries, the altsrc tag can remap any file name to a different managed name. For example, the /etc/hosts file on Linux and the /etc/inet/hosts file on Solaris are the same; they're just located in different places. Managing both separately could easily lead to variations between the two, so instead the altsrc tag can redirect requests for one file to the contents of the other.

Figure 6.8 shows this option in action in a Bundle, telling the Cfg plugin to hand out the contents of Cfg/etc/hosts/ when a client asks for /etc/inet/hosts. This option can be used in many similar cases to keep a repository clean and manageable.

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

Figure 6.8: Using the altsrc tag option

For this task, we can use the altsrc option to remap all of the unique network configuration files generated by the Structure template to a single configuration file template, putting the last piece of the complex puzzle in place. Figures 6.9 through 6.12 show the details of the pieces needed. First, we have the getmacs Probe in Figure 6.9.

```
#!/bin/sh
/sbin/ifconfig -a | grep eth | awk '{print $5}'
```

Figure 6.9: A Probe to collect MAC addresses

This simple script is stored in Probes/getmacs and will print a list of MAC addresses when run on a GNU/Linux machine. As a probe, it will collect the list needed for the Structure template. Figure 6.10 shows the complete Structure template for the task.

```
<Bundle name='networkinterfaces' xmlns:py="http://genshi.edgewall.org/">
    <?python
       files = metadata.probes["getmacs"].lower().split("\n")
       #if 'login' not in metadata.groups
       #   files.pop(0)
    ?>
    <ConfigFile py:for="file in files" \
       name="/etc/sysconfig/network/ifcfg-eth-id-${file}" \
       altsrc="/etc/sysconfig/network/ifcfg-eth-id-MAC"/>
</Bundle>
```

Figure 6.10: The Network Structure template

When a client including the networkinterfaces Bundle asks for its configuration, the server will use the above Probe to generate a Bundle full of ConfigFile entries for each client network interface. Each of these entries includes a MAC address in its file name, and each uses the altsrc option to remap the file to the ifcfg-eth-id-MAC template. The ifcfg-eth-id-MAC template, the final piece of the task, is shown in Figure 6.11.

The first two-thirds of this file is just a block of Python code that parses the network-interfaces.conf file, which is shown in Figure 6.12.

This template is called once for each ConfigFile produced by the above Structure template, with the current file name available to the template in the variable self.path. Here we just split off the MAC address portion of the file name, generate a dictionary of interface information from the configuration file, and insert the correct information into the correct lines in the template. This template file, of course, is placed in TCheetah/etc/sysconfig/network/ifcfg-eth-id-MAC/template.

With all of these pieces in place, we are ready to examine the exact chain of events that happens when a client requests its configuration.

1. The client runs, contacting the server for data.
2. The server replies with the getmacs Probe for the client to run.
3. The client runs the Probe, returning the output to the server.
4. The server uses the Probe data to generate the networkinterfaces Bundle.
5. For each ConfigFile entry in the networkinterfaces Bundle:
    a. The server uses the altsrc option to remap the real file name to an internal name.

```
<%
mac = self.path.split("/")[-1].split("-")[-1]
macs2ips = {}
macs2masks = {}
macs2mtus = {}
tfile = open("/var/lib/bcfg2/etc/networkinterfaces.conf").readlines()
startmode = 'auto'
for line in tfile:
    if(line.startswith('#')):
        continue
    splitline = line.lower().split()
    if len(splitline) < 3:
        continue
    macs2ips[splitline[0]] = splitline[1]
    macs2masks[splitline[0]] = splitline[2]
    macs2mtus[splitline[0]] = splitline[3]
if not macs2ips.has_key(mac):
    macs2ips[mac] = ''
    macs2masks[mac] = ''
    macs2mtus[mac] = ''
    startmode = 'off'
%>BOOTPROTO='static'
ETHTOOL_OPTIONS=''
IPADDR='$macs2ips[mac]'
MTU='$macs2mtus[mac]'
NETMASK='$macs2masks[mac]'
NETWORK=''
REMOTE_IPADDR=''
STARTMODE='$startmode'
USERCONTROL='no'
```

Figure 6.11: The TCheetah/etc/sysconfig/network/ifcfg-eth-id-MAC template

```
# MAC                IP           Mask           MTU
#
# fileserver1.example.com
00:14:5E:5A:92:06    10.10.1.1    255.255.0.0    9000
# www.example.com
00:14:5E:5A:95:D5    10.40.1.1    255.255.0.0    1500
```

Figure 6.12: The network interface configuration file

    b. The server uses the template located at that internal name to generate a network interface configuration file from the server-side configuration stored in etc/networkinterfaces.conf.

6. The server passes the full configuration down to the client.
7. The client installs its configuration as it normally would.

After the work has been done to put this infrastructure in place, keeping it up-to-date is a much simpler task than had we used individual static files for every machine; adding an extra machine is as simple as adding its information to the networkinterfaces.conf file, and, if an extra option needs to be added to all network configurations, there is just one template file to update.

7. The Bcfg2 Reporting System

Unfortunately, many people assume that automated processes are working if they don't hear otherwise. This is not always the case. To combat this problem, we have implemented a reporting system that describes the current operational state of each client and higher-level views of overall deployment state.

As we described in Chapter 2, the Bcfg2 client collects a large amount of data. This data is the basis for the Bcfg2 reporting system. In this chapter, we will describe that reporting system, beginning with the data collected. We follow this with a quick overview of the bcfg2-reports tool. Finally, we describe a day in the life of an administrator using the Bcfg2 reporting system.

7.1 Reporting System Architecture

As we discussed in Chapter 2, the Bcfg2 reporting system houses data collected by the Bcfg2 client. When the client finishes making changes, it generates a report describing its current state and modifications performed by the Bcfg2 client during its operation. This data is sent to the Bcfg2 server, which, in turn, records it in the reporting system database.

7.1.1 Data Collection

A large variety of data is stored by the Bcfg2 reporting system. All of this data is collected by the Bcfg2 client and uploaded to the Bcfg2 server as a part of the normal configuration process. As we mentioned earlier, this data is stored in a relational database. The following is a list of the data stored by the reporting system.

- Bad entries: Bad entries are the most important piece of data stored by the reporting system. Detailed information is stored for each piece of configuration data on a client that explicitly disagrees with one of Bcfg2's configuration goals for that client. Each of these records includes information about the mismatch between the configuration goal and current state. For example, if an incorrect package version is installed, the two versions (expected and actual) are included in the record. Similar data is stored for other entry types.
- Extra entries: The Bcfg2 client discovers client configuration that is not included in the server-specified configuration goals. For each of these, a record is generated that includes as much data as is possible. For example, extra package entries describe the version of package installed.
- Modified entries: Each configuration entry that is modified during Bcfg2 client operation is recorded. A description of the complete operation, including

initial and final states, is available. This information is comparable to the data stored for bad entries; the only difference is that the data describes initial and final states instead of initial and desired states.

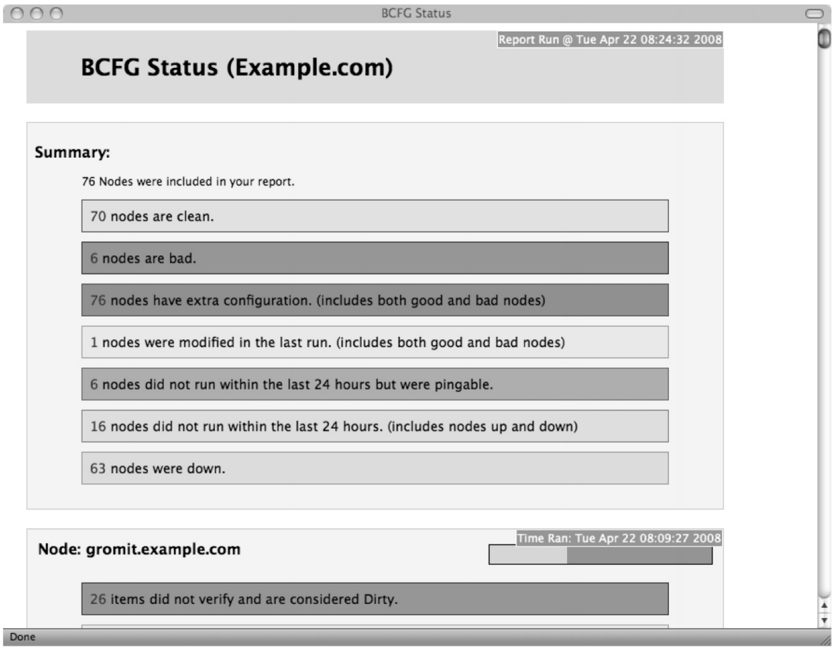
7.1.2 The bcfg2-reports Tool

bcfg2-reports is a command-line tool that queries the reporting system database. It can be used to show all current client states, or can query clients by a variety of criteria. This tool is a workhorse for monitoring the current state of all clients' configuration state. bcfg2-reports will play a major role in each of the examples later in this chapter.

bcfg2-reports can be used in two different basic modes. One returns information about a particular client. The other finds clients based on reporting system criteria. As you might expect, the latter mode is much more useful, because you can find unexpected information with it. Running it with -h will print usage information. In its basic modes, it can query all clients, clients by status (clean or dirty), or a particular client. It can also display clients based on individual or combined bad or extra entries. These later options enable cron jobs that check for clients that are misconfigured for particularly important entries.

7.1.3 Web Reporting Front End

The Bcfg2 reporting system also has a Web-based front end. It provides a set of basic views of system-wide status information, including summaries of client and dirty clients, clients that haven't checked in recently, and so forth. Each of these views is linked to detailed per-client information.


BCFG Status

Report Run @ Tue Apr 22 08:24:32 2008

BCFG Status (Example.com)

Summary:

76 Nodes were included in your report.

70 nodes are clean.

6 nodes are bad.

76 nodes have extra configuration. (includes both good and bad nodes)

1 nodes were modified in the last run. (includes both good and bad nodes)

6 nodes did not run within the last 24 hours but were pingable.

16 nodes did not run within the last 24 hours. (includes nodes up and down)

63 nodes were down.

Node: gromit.example.com

Time Ran: Tue Apr 22 08:09:27 2008

26 items did not verify and are considered Dirty.

Figure 7.1: The Bcfg2 Web reporting front end

Figure 7.1 gives a small glance at the Web front end. The title box displays one line with the date and time that the report was run and a large title showing the name of the Bcfg2 system being reported on, helping administrators of several disconnected networks keep track of which network they are currently looking at and how old the data is. The summary section displays a number of expandable boxes that show how many machines are in what state. In this case we are doing all right: out of 76 machines, only 6 do not completely match up to their Bcfg2 images. Below the summary box is an individual report for every machine the server knows about, with bad machines showing up first. This section of the report shows which exact items are wrong, making it easy to review items that need to be updated on the client.

7.2 Use Cases

The simplest use for the reporting system is to provide an up-to-date overview of Bcfg2 and its operations. The Web reports give a nice high-level view of what the overall states of clients are. Any environment will have a normal overall state; when this changes it is a good sign that something unexpected is going on. For example, when a lot of clients are in dirty state, that is a good indication that something is going wrong; either the desired state is not reachable or the client isn't being given the opportunity to make changes. A rapid increase in extra entries can signal manual administration activities. All of these activities are reactive; that is, administrators notice something that doesn't look correct and go figure it out.

Once this routine is established, administrators will have a set of conditions that signal real problems. This likely corresponds to areas of either rapid configuration changes, important configuration, or security concerns. From this list of concerns, administrators can use `bcfg2-reports` to build a set of cron jobs that detect when these bad conditions occur.

Above all, the reporting system is meant to be a tool that makes the configuration deployment process much more transparent than it normally is. The reporting system watches over what is going on in the background, making it easy to see what changes have happened and what problems have occurred. Without it, administrators need to either confirm on a regular basis that every file they want pushed out to a set of clients is out there (very time-consuming and unlikely to happen often) or just assume everything is working correctly and wait for problems to arise (dangerous). The reporting system is one of Bcfg2's hidden gems that makes adoption and everyday use of the tool much easier.

8. What Next?

Having gotten this far, readers should have a good foundation from which to deploy Bcfg2: the ability to automate common configuration tasks and provide robust enforcement of configuration policies. This leaves the actual, and potentially arduous, deployment of Bcfg2. In this chapter we present some tips for easing and accelerating this process, and we also point out additional information sources for Bcfg2.

8.1 Deployment Tips

Deploying any system management tool can be difficult in a sophisticated environment, and Bcfg2 is no exception. The following tips will simplify your deployment and speed the process along.

- Start small: Begin by managing limited aspects of a small number of client systems. This allows you to get used to the way that Bcfg2 works in a controlled environment.
- Start simple: Bcfg2 allows both simple and complex representations of configuration. Advanced techniques such as templating are quite powerful, but introduce additional levels of debugging after template development is done. Save the more complex configurations for the second day.
- Set up the reporting system early: The Bcfg2 reporting system is a potent tool for understanding the operations of Bcfg2 across a network. Deploy it as soon as possible.
- Master the server-side query tools early: Tools like `bcfg2-info` can provide great insight into the functioning of the Bcfg2 server and configuration rules.
- Ease into deployment using dry-run mode: Begin by deploying Bcfg2 in dry-run mode, where changes are not automatically performed. This provides an added opportunity to understand the results of making configuration rule changes for the overall environment.
- Follow an organized deployment plan: During the conversion from manual administration to centralized configuration management, someone will make changes on a client system, only to have them overwritten by automated configuration processes. Good communication about which aspects of configuration are managed across clients can prevent this from occurring.
- Get group buy-in early: Alterations to configuration management procedures impact the most basic aspects of system administration. It is well worth the time to get everyone on board with the changes and properly trained to interact with Bcfg2.

8.2 Further Reading

This booklet, while providing an overview of common areas of Bcfg2 functionality, is necessarily incomplete. Bcfg2 is under active development and frequently incorporates new features. There are several places where up-to-the-minute information on new Bcfg2 features can be attained:

- Web site: http://trac.mcs.anl.gov/projects/bcfg2
- IRC channel: irc.freenode.net #bcfg2
- Mailing list: bcfg-dev@mcs.anl.gov

Appendix A. Bcfg2 XML Options

Bcfg2 uses XML files as configuration rules. Each of the entry types used in the system has different options, useful in different circumstances. First, in Section A.1, we describe the overall processing model used by the Bcfg2 server. In Section A.2 we describe each of these entry types in turn, including all required and optional attributes. In Section A.3, we document the formats used to control XML-based server plugins. XML was chosen specifically for its ease of generation; this chapter is particularly useful for users autogenerating configuration rules.

A.1 Configuration Processing

As we described in Section 2.2.4, the Bcfg2 server builds client goals using a process broken down into three steps. The first is client metadata resolution. The second is the construction of abstract configuration goals. Recall that these goals have entry types and names, but no client-specific entry data is included. The result of this process is a series of lists of abstract configuration goals. Finally, each entry in this list is "bound" using the client's metadata. This last step places client-specific goal information into the client configuration. This set of literal configuration goals is served to the client system.

In general, this process is quite straightforward; when presented with an abstract configuration goal, the server determines which server plugin has data for that goal and routes the bind request to that plugin, which deposits client-specific data in the abstract goal, rendering it a "bound" goal.

Two factors can affect this process. Bound goals are goals that have been placed in the abstract configuration with client-specific data already included. altsrc attributes modify how the bind process works for a given goal.

A.1.1 Bound Goals

Bound goals are abstract configuration goals that include literal, client-specific data. They are created by including an entry that has the prefix Bound added to a normal goal. When the goal binding process encounters a bound goal, the binding process consists of removing this prefix from the goal tag; no other action is taken.

Bound goals are typically useful in two cases: (1) if a goal is completely static and universal to all clients, it can be useful to specify it in one place and never process that goal; (2) when using SGenshi to build templated abstract configuration structures. Use of SGenshi is discussed in Section A.3.5.

A.1.2 Use of altsrc

altsrc attributes are a way to tell the Bcfg2 server to bind a goal as if it had a different name. It is activated by adding the altsrc attribute to an abstract goal. When the server binds the goal, it temporarily resets the goal name during binding and resets it to the original name afterward.

This can be useful in a variety of cases. For example, some configuration files have different names across platforms while retaining the same format. In this case, it is convenient to manage a single repository in Cfg for that configuration file.

altsrc attributes are also useful with generic templates. In many cases, a single template can produce several different files. Templating logic can detect both the file name being produced and the non-altsrc name for the entry, so that the proper file contents can be produced.

Finally, altsrc attributes are useful when interacting with Pkgmgr virtual package targets. These are described in more detail in Section A.3.2.

A.2 Goal Types

This section describes each of the goals that can be managed by Bcfg2. The precise format described here is used in two ways by the server. Configuration goals sent to clients contain a series of these, with many of these attributes included. These definitions guide the behavior of the Bcfg2 client.

The other use for these goal descriptions occurs in the configuration rules specified on the server. Recall that rules describe a potential end state for a given goal that applies to a client or group of clients. The goals are frequently described in this format.

A.2.1 Actions

Actions are configuration entries that execute commands when the specified conditions occur. Action prerequisites are tied to entries in a Bundle, hence actions can only be used inside Bundles. Actions can also be used as a prerequisite to installation of entries in a Bundle. Unless exit status is ignored, a failing pre-action will prevent modification of entries in the enclosing Bundle to be performed; all entries included in that Bundle will not be modified. Similarly, failing actions are reported via the reporting system, so they can be centrally observed. All action entry attributes are described in Table A.1. Actions are typically defined using the Rules plugin.

Name	Description	Values
timing	When the action is run	pre, post, both
name	Action name	String
command	The full command to be run	String
when	Under what circumstances the action will run	always, modified
status	Should the return code be reported?	ignore, check

Table A.1: Action attributes

A.2.2 ConfigFile

ConfigFile goals describe configuration files on the filesystem. The paranoid attribute controls client behavior; when it is specified, the client retains a backup copy of the file whenever it is modified. All of the other attributes describe the goal state of the managed configuration file. These attributes are described in Table A.2. File contents are included as a text node in the XML element.

Configuration files aren't typically managed using this XML representation. While this format can be used either in the Rules plugin or by specifying BoundConfigFile goals in a structure plugin, several more natural representation formats are provided by the Cfg, TCheetah, TGenshi, and SSHbase plugins. The first three of these are general-purpose plugins, while the latter only manages SSH keys.

Name	Description	Values
name	Path to configuration file	String
perms	Permissions of the file	String
owner	Owner of the file	String
group	Group of the file	String
encoding	Contents encoding	(ascii/base64)
paranoid	Save a file copy upon installation	(true/false)

Table A.2: ConfigFile attributes

A.2.3 Packages

Packages are configuration entries corresponding to installed software packages on clients. When included in Bundles, package entries perform a Bundle-aware verification. In this case, packages ignore verification failures for any entries also contained in the Bundle. Due to the large range of features offered by underlying package management, package entries have the largest number of optional attributes. These are described in Table A.3 (next page). A majority of the options control client installation and verification. However, several of the options (file, srcs, and simplefile) control the behavior of built-in functions of the Pkgmgr plugin, where package entries are typically described.

The name attribute describes the name of the package; likewise, the version attribute describes the version. multiarch describes which architectures the package should be installed for on x86_64 Red Hat–like systems. reloc governs package relocation on RPM-based systems.

The RPMng driver has the concept of package instances, as several packages of the same name can be installed at once if they differ in version, architecture, or other attributes. The options available to the instance tag are described in Table A.4 (next page).

A.2.4 Permissions

Permissions entries control the permissions of filesystem entities. These attributes are described in Table A.5 (next page). Permissions entries are typically managed by the Rules plugin.

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

Table A.3: Package attributes

Name	**Description**	**Values**
simplefile	Package file name	String
epoch	Package epoch	Numeric
version	Package version	String
release	Package release	String
arch	Package architecture	String
verify_flags	Comma-separated list of rpm verify options. See the rpm man page for details.	String
pkg_verify	Do the rpm verify	true/false
install_action	Install package instance if it is not installed.	install/none
version_fail_action	Upgrade package if the incorrect version is installed.	upgrade/none
verify_fail_action	Reinstall package instance if the rpm verify failed.	reinstall/none

Table A.4: Instance attributes

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

Table A.5: Permissions attributes

A.2.5 Directory

Directory entries control the permissions, contents, and ownership of directories. Attributes are described in Table A.6. These entries are typically managed by the Rules plugin.

Name	Description	Values
name	Directory name	String
perms	Permissions of the directory	String
owner	Owner of the directory	String
group	Group owner of the directory	String

Table A.6: Directory attributes

A.2.6 Symlink

Symlink entries manage filesystem symlinks. Symlink has a single attribute, to, which describes its target (see Table A.7).

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

Table A.7: Symlink attributes

A.3 Plugins

Several plugins use XML files as input. Structure plugins uniformly produce abstract goal entries with a tag corresponding to the goal types listed in Section A.2. As described in Section A.1, the altsrc attribute and bound goal modification are also usable.

Most server plugins that use XML input files provide a selection mechanism for describing rules in increasing priority. This mechanism uses group and client elements to describe the scope of enclosed goal descriptions. Group and client elements work in the expected manner: group clauses match for clients that are members of the named group, and client clauses match for the named client and no other. Nested group and client clauses are conjunctive; nested scopes apply only in the case when all enclosing clauses match. When no matching clauses are used, the scope applies to all clients. Clauses can also be negated through use of the negate attribute.

In the Rules and Pkgmgr plugins, multiple files can define rules that overlap with one another. Each file is labeled with a numeric priority; when this overlap condition occurs, the file with the highest priority is used.

A.3.1 Rules

The Rules plugin can manage goals of any sort. All goals are enclosed in a Rules XML element, which has a single priority attribute. Inside this Rules element, both scope clauses and goals of any sort can be included.

A.3.2 Pkgmgr

The Pkgmgr plugin is used to manage package goals. It works in a similar fashion to the Rules plugin; the Pkgmgr plugin adds package-specific functionality. The PackageList tag is the root element of Pkgmgr input files. It is used to set parameters for all package goals included within it. Download URLs, file priority, package type, and multiarch information are all specified in this way. Each is described in Table A.8.

Name	Description	Values
priority	Priority	Integer
url	Download location	String
file	Package file name. Several other attributes (name, version) can be automatically defined based on regular expressions defined in the Pkgmgr plugin.	String
multiarch	Comma-separated list of the architectures of this package that should be installed	String
srcs	File name creation rules for multiarch packages	String
type	Package type (rpm/yum/deb/encap/sysv/blast/portage/freebsd)	String

Table A.8: PackageList attributes

Package descriptions, described in detail in Section A.2.3, are contained inside the PackageList element. Group or client selection clauses can also be used.

A.3.3 Base

Base is a plugin used to list all base configuration elements for your site. These are most often the elements that are installed by default by a basic operating system installation: the libc package, the package that creates entries in /dev, and other independent operating system packages are good examples. Note that these are independent—unlike Bundler entries, an update to an element in Base will never affect another element. Base is likely to hold more packages and independent configuration files than all of your Bundler entries combined, since it is a good place to define any pieces that just need to be installed without further configuration.

A.3.4 Bundler

The Bundler plugin lets you define dependent configuration elements and bundle various elements together into discrete building blocks. An example of dependent configuration elements are the NTP package, service, and configuration files: if the NTP package or configuration files are updated, the client should restart the NTP service to make sure the changes take effect. Similarly, if the NTP package is updated, the client should recheck the configuration file to make sure it wasn't overwritten by the new package. Bundles can be very simple, as seen in Section 4.1, complex, or even generated at client run time. Chapters 4 and 6 provide several examples of how Bundles are used.

A.3.5 SGenshi

The SGenshi plugin is used to generate XML structures on the fly. This can include Bundles, Pkgmgr lists, or any other XML structure that the Bcfg2 server uses. This plug-in uses the Genshi templating engine to do its heavy lifting, making it a very powerful tool to use when static configurations don't do everything you need. An example of using SGenshi is discussed in Section 6.3.

Appendix B. Client Identification

When a client connects to the Bcfg2 server, the server needs to identify it, so that it can serve the appropriate configuration to the client. During this process, the server also needs to authenticate the client, so that malicious clients cannot gain access to other clients' data. These topics are intertwined; client authentication is dependent on client identification.

Before a client can be authenticated, its identity must be established. This can be done in two ways. The Bcfg2 server's default mechanism uses the DNS resolution to establish a name for the client. This approach works well in most cases and requires no manual setup. In settings where clients are mobile or behind NAT, a different approach is required.

The alternative is the use of a per-client UUID. This UUID is used as the user name in HTTP basic authentication and is unique to the client; hence, it can be used to differentiate between clients behind NAT or mobile clients.

Once a client has been identified, it must be authenticated. This can be accomplished through any of several mechanisms. Bcfg2 can be configured with a global password. This is convenient in cases where unattended system bootstrapping is required—clusters and the like. Alternatively, per-client passwords can be specified as well. These mechanisms can be intermixed; some clients can use a global password, while more sensitive systems can use a unique password. Clients can also be configured with a password but downgraded into insecure mode, where the client can authenticate with either its password or the global password. Typically, systems will be toggled into insecure mode for system bootstrapping; once the systems are built, they are reset into secure mode. All transactions between the client and server occur over an SSL-encrypted socket; SSL is used as content protection for the transaction. Finally, IP address restrictions can be imposed on clients; clients will be denied unless they connect from an appropriate IP address.

The authentication approach used by the Bcfg2 server has been designed to provide maximum flexibility to administrators. In some situations, security is valued above all else. In other cases, compromises must be struck for expediency's sake. The mechanisms described above provide options to system administrators to tailor authentication to site-specific priorities.

About the Authors

Narayan Desai is a researcher in the Mathematics and Computer Science division of Argonne National Laboratory. His work centers on the management and operations of large-scale HPC and experimental systems. In addition to Bcfg2, he works on scheduling, fault tolerance, and high-performance networking issues.

Cory Lueninghoener is a senior high-performance computing system administrator with Argonne National Laboratory's Leadership Computing Facility, where he helps keep very large computers running smoothly. His interests are currently focused on methods for running large-scale computing resources with unusually small groups of people. When not advocating and implementing configuration management strategies, he contributes design and code to the Bcfg2 project.