Booklets in the Series

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

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

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

#14: System Configuration
Paul Anderson

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

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

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

#10: Budgeting for SysAdmins
Adam Moskowitz

#9: Backups and Recovery
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
John Sellens

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

#5: Hiring System Administrators
Gretchen Phillips

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

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

#2: A Guide to Developing Computing Policy Documents
Edited by Barbara L. Dijker

17 Short Topics in
System Administration
Jane-Ellen Long, Series Editor

LCFG: A Practical Tool for System Conf guration

Paul Anderson

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 978-1-931971-61-4

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

Acknowledgements v

1. Introduction 1
- 1.1 This Booklet 1
- 1.2 Approaches to System Configuration 2
- 1.3 What Exactly Is LCFG? 4
- 1.4 Is LCFG for You? 5
- 1.5 The LCFG Software 7

2. A Quickstart Tutorial 8
- 2.1 Installing LCFG (Or Not) 8
- 2.2 Trying It Out 9
- 2.3 Beyond the File Component 11
- 2.4 Writing Custom Components 13
- 2.5 Where Next? 16

3. Writing and Compiling Configurations 17
- 3.1 LCFG Configuration Files 18
- 3.2 The LCFG Language 20
- 3.3 The LCFG Server 29

4. LCFG Components 34
- 4.1 The Anatomy of a Component 34
- 4.2 Some Common Components 37
- 4.3 More Components 40
- 4.4 Updating Software 41

5. Managing a Site with LCFG 44
- 5.1 How Much to Automate? 44
- 5.2 Who Manages What? 45
- 5.3 Managing Change 46
- 5.4 Installing from Bare Metal 47

6. Writing Components 49
- 6.1 Schema Files 50
- 6.2 The Component Framework 57
- 6.3 Testing 70
- 6.4 Packaging and Installation 72

7. Finally . . . 74
- 7.1 LCFG Developments 74
- 7.2 The Future 74

Appendix A. Bootstrapping an LCFG Installation 76

Appendix B. Buildtools 78

Appendix C. The Linux Installroot 82

Index 85

About the Author 89

Figures

1.1. The LCFG Architecture 4

3.1. Standard Mutation Macros 23

3.2. Summary Page 32

3.3. Individual Client Display 32

4.1. Logserver Display 40

6.1. Standard Validation Macros 51

Acknowledgements

LCFG has been around for over 10 years now, and very many people have been involved in the development, either with code, associated research projects, or just useful ideas and feedback. Alastair Scobie, George Ross, Kenny MacDonald, Stephen Quinney, and Simon Wilkinson have all made significant contributions to the code, and many more people have contributed to various components.

I am particularly grateful to Stephen Quinney and Kenny MacDonald for allowing me to use their tutorial examples in this booklet, and for their advice on presenting LCFG to new users. Thanks also to Jane-Ellen Long for editing and to everyone else who read draft copies and provided such helpful suggestions.

1. Introduction

Even a site with a small network of computers is likely to have hundreds of configuration files.[1] The content of these files determines how the overall system works and what it does; this is what makes the difference between a lab of student machines and the same set of hardware providing a Web service.

Working out what to put in the configuration files takes a lot of skill and experience. Any mistakes are likely to cause some kind of failure or security breach. A system administrator will usually be responsible for setting these files up correctly and updating them as the system and the requirements change. This is *system configuration*.

For a site of any significant size, this is usually too difficult or hazardous to do completely by hand. LCFG is one of several tools which have been developed specifically to help automate the configuration process.

LCFG is not for everybody. It takes a lot of commitment and effort to adopt. It requires an ongoing maintenance effort, and it works best where there is a particularly disciplined approach to site management. However, the rewards can be huge; your configurations are much more likely to be correct (and hence secure) and up-to-date. Maintaining them will take less effort, and that effort can be less skilled. Even extreme changes to the site configuration will be much easier: people often find that LCFG-managed configurations are in a constant state of flux, but this is simply because LCFG is able to respond instantly and reliably to changes in requirements.

1.1 This Booklet

This booklet is a practical guide to using LCFG for managing real-world site configurations. The rest of this chapter talks about the architecture of LCFG and explains some general principles to help you decide whether it is appropriate for your situation. If you are impatient to see what LCFG feels like in practice, you can go straight to Chapter 2 for a basic demonstration (but you will probably want to come back and read this later). The remaining chapters go into more detail about LCFG and its use:

- Chapter 2 is a “quickstart” tutorial which uses an image pre-installed with LCFG to provide a hands-on introduction.

1. I tend to talk about “configuration files” because these are the most common way of storing configuration information on UNIX systems—but the the problems are the same even when this information is stored in databases or behind GUIs.

- Chapter 3 describes the simple format used to specify configuration parameters and talks about how these parameters are manipulated by the LCFG *compiler*.
- Chapter 4 explains the concept of LCFG *components* and introduces some common examples. These are the small scripts that take the configuration descriptions and turn them into specific OS configuration files.
- Chapter 5 covers the practical issues of using LCFG to manage a real site.
- Chapter 6 describes how to write custom LCFG components.
- Chapter 7 presents a few concluding thoughts.

LCFG is still evolving, and the details of the implementation will certainly change. The main features described in this booklet are likely to remain stable, but full documentation is included with the latest software, which is available from http://www.lcfg.org. This site also contains slides and recordings of live LCFG tutorials, as well as some research papers and a pointer to the LCFG mailing list.

This booklet does not include any in-depth discussion of the system configuration principles behind the software. If you are interested in this—and it does help to explain *why* LCFG does things in certain ways—then you might like to read my SAGE booklet *System Configuration*.[2]

A Word About Terminology

The machine running the LCFG server code is often just called the *server*, but I have tried to use *LCFG server* whenever there is any ambiguity, to distinguish it from a machine running some arbitrary service. The term *LCFG server* is also used to refer to the code itself, but whether code or machine is meant should be clear from the context; similarly for the term *client*.

1.2 Approaches to System Configuration

Unfortunately, system configuration is still a developing subject, and there aren't any clear standards or widely recognised approaches; most people start by gradually evolving their own scripts and tools, based on their original manual procedures.

This approach has a lot of advantages; people can gradually adopt an automated solution, learning the tools and principles as they go along. There is no "big bang," and everyone can be reasonably confident that the system is going to continue to work in the same way as before. The sorts of tools best suited to this approach are usually based on familiar principles (e.g., UNIX shell programming)—or at least, they specify configurations in terms of familiar concepts such as processes and configuration files (e.g., Cfengine[3]). These tools don't involve a radical change in the way people need to think about their systems, and they don't involve a steep learning curve.

2. http://www.sage.org/pubs/14_sysconfig/.
3. http://www.sage.org/pubs/16_cfengine/16_cfengine.html.

There is a problem, however, with this incremental approach. Once the mass of configuration details has been tamed, the higher-level conceptual problems start to become more apparent: How can you smoothly manage constantly changing configurations so that services are not disrupted? How can you make sure that different people can work together on the configuration of a site without conflicting with one another? How can you be certain that your configuration is always consistent—that clients are always configured in a way that matches their servers? Can you automatically analyse your configuration to identify any unexpected interactions that might allow an attacker to compromise the system? Can you support "autonomic" recovery so that things reconfigure themselves automatically in the event of a failure?

The types of tools mentioned above are difficult to use in a way that supports this kind of higher-level thinking. For example, most people who write Cfengine (or shell) scripts to configure their firewall and their Web server don't naturally write them in a connected way so that the firewall is automatically updated when the Web server is moved to a different machine. Other tasks, such as extracting explicit information about service dependencies, would be even harder (this is very useful if you want to automatically analyse single points of failure, for example).

Our own in-house installation consists of about 1200 machines (200 of which are servers of some kind) with very diverse configurations. For a long time, the configuration has been fully automated and "prescriptive"[4]—we make no manual configuration changes on production machines, and we like to think of our configurations as extremely reliable and consistent. Recently, we have been interested in tools and procedures to help manage these kinds of higher-level concerns; this has spawned a few research projects, as well as a lot of practical discussion and experimentation.

LCFG[5] is the configuration tool we developed to support this environment. It has evolved over 15 years and currently goes further towards addressing some of the high-level issues than any other tool that we know of. LCFG uses a very simple and consistent format for writing down configuration parameters, and the complete configuration of all the machines is usually held centrally on one LCFG server. This means that we can easily use other programs to analyse the configuration of the whole site, or even generate some aspects automatically. This is what allows us to check the consistency of configurations between clients and servers, make staged releases of configurations across the whole site, or quickly reconfigure new machines as replacement servers when one fails.

4. *Prescriptive* means "of, or relating to the imposition or enforcement of a rule or method." In the past, we have sometimes mistakenly used the term *proscriptive*, which has a different meaning.

5. The name LCFG was originally derived from "Local ConFiGuration system."

1.3 What Exactly Is LCFG?

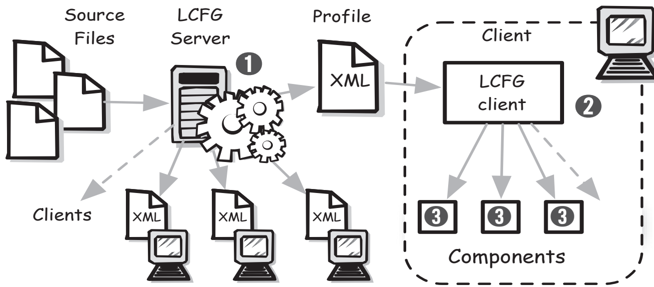


Figure 1.1: The LCFG Architecture

There are three main parts to the LCFG software (see Figure 1.1):

- The LCFG *server*. This collates all of the configuration information from the *source* files and creates a single XML file (the *profile*) for each machine. The profile contains all of the configuration parameters (*resources*) for that machine. The profile is exported to the LCFG client using a standard Web server such as Apache.
- The LCFG *client*. This runs on every machine. It downloads a new profile for the machine whenever the configuration changes.[6] The LCFG client works out which *components* are affected by the change and calls the corresponding script for each one. The client can be configured to return a simple acknowledgment to the server so that it can keep track of any systems that fail to reconfigure correctly.
- The LCFG *components*. Each machine includes a collection of scripts called *components*. These are responsible for translating the LCFG resources into machine-specific configuration files. Each component is responsible for a self-contained subsystem—for example, the password file or the Apache server. The component generates the configuration file and takes care of other low-level details such as restarting any associated daemons.

All of the interesting work happens in the LCFG server. There is no one-to-one correspondence between the source files and the profiles; the profile for a machine is created by assembling information from many source files according to the classes assigned to that machine. For example, a particular machine may be a “Dell GX250” running a “Web server” in “building X.” The LCFG server collates and merges all of these resources, managing any conflicts and prioritisation. This is what allows these different *aspects* to be managed by different people.

In addition to simply merging resources, the LCFG server can perform some more complex operations. For example, it can collate parameters from many machines and present them as part of the configuration for some other machine; this could be used to

6. The LCFG server sends a short notification to the LCFG client when a change occurs. The client also polls the server regularly in case it misses a notification.

collect the IP addresses of all the Web servers to include as part of the configuration for the firewall machine (this is called a *spanning map*).

The LCFG configuration is truly *declarative*. This means that it specifies what the configuration should look like rather than specifying what changes to make. The SAGE *System Configuration* booklet explains why this is a very important property. For example, it allows an LCFG client safely to miss configuration changes or receive the same changes multiple times—the configuration is simply synchronised with the most recent profile, and it is always correct.

In addition to maintaining the configuration of an existing system, LCFG supports a number of techniques for installing new systems from bare metal, preconfigured according to their LCFG specification.

1.4 Is LCFG for You?

Configuring a modern computing installation is complicated. Good configuration tools are like compilers in that they will help you translate your overview into a detailed implementation. But the real work lies in developing the specifications for your installation. It is not possible to take LCFG (or any other tool) out of the box and expect it to create a complete, functional site; it needs careful thought and effort to develop specifications appropriate to your needs and to express them in the appropriate language.

Adopting or changing the configuration tool for a large site is also a big commitment; people need to learn new technologies and procedures. They need to gain sufficient confidence in the system to trust it with vital services, and any new system is likely to remain in place for a long time.

Unfortunately, most sites probably develop their configuration strategies gradually, without ever making any conscious decisions based on the kind of high-level issues we have been discussing. By the time the site configuration is mature enough for these problems to become apparent, there is usually a huge investment in the existing systems.

If you are thinking about adopting a new configuration technology, you should probably spend some time evaluating the available tools against your own particular requirements. The SAGE *System Configuration* booklet provides a good background to the factors you may want to consider. Here are a few key points about LCFG:

- LCFG is designed to separate the configuration problem into two main layers. The configuration of the site is represented in a simple, uniform language. This forms a solid foundation for other processes (manual or automatic) to manipulate the whole site configuration. The components translate this into the appropriate details for particular configuration files.
- LCFG is not simply for managing desktop machines (although it is very good at this). That work often just involves slight variations on similar configurations (such as different sets of software packages). The advantages of LCFG

become more obvious when managing servers, which tend to have more diverse configurations with complex relationships among them. We manage over 200 servers running interrelated services such as DHCP, DNS, NFS, AFS, SMB, NTP, IMAP, SMTP, SSH, and HTTP. This is fairly typical for a large site.

- LCFG is particularly effective for larger sites where there is a mixture of skills and specialists collaborating on the management of the whole fabric. It breaks down the configuration task into at least three levels: domain specialists (e.g., Mr. Networking), who write component code and perhaps Web forms; more general system administrators, who write macros encapsulating configuration options for various services; and technicians, who assign these sets of configuration parameters to specific machines. LCFG helps these people to work together without conflicting.
- This LCFG model is different from the traditional one, in which individual system administrators often have responsibility for all aspects of a particular machine; our devolved approach becomes necessary as soon as configurations reach sufficiently specialisation that one person can no longer be expected to understand all of the implications.
- LCFG ships with a large number of standard components for common subsystems such as Apache, DNS, and Kerberos. These components translate the LCFG parameters into the corresponding configuration files. This means that a lot of things can be configured simply by providing parameters to standard components. Almost every site will have some special requirements, though, and some editing or creation of local components will almost certainly be necessary. The amount of work is usually small, but it does require coding ability, and it takes time to understand the framework.
- The components are written within a simple, standard framework. This means that they can be shared between sites much more easily than arbitrary scripts.
- LCFG can be used as a prescriptive tool, i.e., to manage every aspect of a machine so that there is no manual intervention (or configuration by other tools). There are lots of advantages to doing this, and the SAGE configuration booklet covers the topic in some depth. But this does not have to be the case—LCFG can easily be used to manage just a few configuration files. The rest of the system can be managed manually or configured using other tools.
- What is important is to avoid a conflict between these approaches. If LCFG is being used to manage some aspect of the system, administrators must trust the tool and not fight it by making manual changes. This sometimes requires discipline and education.
- LCFG configuration parameters are specified in a uniform format so that they can easily be interpreted and generated. These parameters are translated (by *components*) into machine-specific configuration files. In one sense, the LCFG specification is higher-level: it represents more information than the configuration files themselves. This means that it is not really practical to take an exist-

ing system and automatically generate an LCFG configuration for it. To do this, you need to understand the configuration of the machine (the "why" as well as the "what") and manually create an equivalent LCFG configuration.

- A tool which interfaces to so many system components is naturally sensitive to updates and variations between OS versions. Our own site is based largely on Red Hat/Fedora/Scientific Linux, and these are well supported; tested versions of many components are available for various releases of these operating systems. With a good understanding of the framework, porting the core and relevant components to other UNIX-based systems should be fairly straightforward; we are running production systems based on Solaris and Mac OS X, for example. But these are not so well supported or packaged for export, and more commitment would be required to manage these platforms.
- There is no technical reason why LCFG should not be used under other operating systems A demonstration LCFG client has been created for Microsoft Windows, together with components for Windows-specific applications such as Registry modification. However, Microsoft and the Windows world have their own approaches to configuration, and we have never had strong enough need to develop this platform.[7]

1.5 The LCFG Software

A lot of software is available on the LCFG Web site, which can be confusing. This is largely because the Web site includes all of the components that we have developed for various subsystems. You will probably find an existing component for many of the things you want to manage. The software is continually evolving, and we continue to export all of the regular releases. This means that you can continue to download software compatible with the version you are currently running without being forced to upgrade. Finally, LCFG components are often tied closely to the version of the operating system, and we export packages and tested versions for several different operating systems.

The packages are available individually or in bundles:

- The "core" bundle contains the libraries and shared code, the LCFG server and client, and a few basic components. This is a reasonable starting point for experimenting with the system (but see Chapter 2 for the easiest way to get started).
- The "standard" bundle contains the most common components.
- The "optional" and "contributed" bundles contain additional components.

There are also sets of prerequisite packages (e.g., Perl modules) which are not part of LCFG and not normally shipped as part of the OS. Various disk images (e.g., a demonstration system as a VM image) and sets of sample configuration files are also available.

The pre-packaged software is shipped in the native format for the OS, usually RPM. Direct access to the CVS repository is available for building on other platforms.

7. In particular, we run very few Windows servers.

2. A Quickstart Tutorial

For those who like to experiment and learn by doing, it can be difficult to get started with LCFG—it is a large system with a lot of documentation. This chapter is a practical introduction to the tool, using a pre-prepared image. This provides an instant LCFG environment for experimentation, without all the issues of compatibility and installation.

The examples in this chapter are based on tutorials which are available as slides and audio files on the LCFG Web site.

2.1 Installing LCFG (Or Not)

Once LCFG is running, it can update itself without manual intervention, and it can automatically install new machines with LCFG already present. If we want to use LCFG at a new site, though, there is a "bootstrapping" problem: how do we get an initial version up and running?

Appendix A describes how to do this manually—a number of packages and an initial set of configuration sources need to be installed. But if there is no distribution of LCFG built specifically for your target platform, binaries will need to be built and configured, and there may be some compatibility issues. This effort is worthwhile for a committed production site, but probably not if you are simply interested in evaluating the system.

By far the easiest way to get started is to use a virtual machine image. The LCFG Web site contains a VMware image which has been prebuilt to include a minimal set of LCFG software and configuration files. You can use this image immediately with VMware Player to work through the following examples. You can then use it as an LCFG server to bootstrap other physical machines, including new LCFG servers.

You will need about 2.5 GB of disk space to download the complete OS image:

```
$ rsync -av rsync.lcfg.org::vmplayer/lcfgfc6 .
```

In the following examples, you will be making some persistent changes to this image, so you may want to make a backup copy to avoid having to download it again if you later want a clean copy.

VMware Player is available from http://www.vmware.com. Under Mac OS X, you will need VMware Fusion. Currently, VMware Player is free and VMware Fusion is available at small cost.

2.2 Trying It Out

If you run the starter image, it should boot a copy of Fedora Core 6 with the core LCFG software already installed and the necessary daemons automatically started. You should be able to log in as the user lcfgfc6 with the password lcfgfc6. Typing startx will start the window system, and you should be able to obtain a root shell in one of the windows using sudo (use the same password).

The virtual machine is configured to run both an LCFG server and a client, so it is using LCFG to configure itself. If we edit the source file for the local machine's configuration, the LCFG server will recompile the configuration and pass it to the LCFG client. The machine will then reconfigure itself to match the new specification:

Change directory to /var/lcfg/conf/server/source and look at the file localhost. This is the configuration description for the local machine. It is a plain-text file and supports the same syntax as the C preprocessor—you can use C-style comments (not C++) and directives to include other "header" files. The example includes some standard header files with default configuration information—we won't go into details of their contents at this stage. What we are going to do is to change this file to add some additional configuration information:

```
/* lcfg example host source profile */

#include <local/site.h>
#include <lcfg/os/minimal.h>
#include <lcfg/hw/vmware_ws5.h>
#include <lcfg/options/lcfg-server.h>

/* eof */
```

As a simple example, let's use LCFG to manage the contents of the file /etc/motd. This is the "message of the day" file which is displayed when a user logs in. Notice that this file is initially empty. We need to add some configuration "resources" to the source file. Copy the file /root/workshop/part1/example1 to replace localhost in the source directory. This file is heavily commented, but there are only four new lines of real configuration:

```
!file.files          mADD(example)
file.file_example    /etc/motd
file.type_example    literal
file.tmpl_example    Welcome to the LCFG tutorial.
```

This probably seems rather cryptic at this stage, but we are not interested in the details for now. You can read the comments and manual pages later to see exactly what these resources mean. Simply note the uniform syntax, which includes the name of the component and the name of the resource, separated by a period. This is followed by the value for the resource. In this case, we are asking the file component to manage the file /etc/motd (in addition to any other files it is managing) and to place the literal string Welcome to the LCFG tutorial. into the file.

Now look again at the file /etc/motd. It should contain the string Welcome to the LCFG tutorial. Notice that this has been reconfigured without running any explicit commands—all we did was to edit the source file!

So What's Going On?

Behind the scenes, this simple file change has involved quite a long sequence of events. We are going to follow the process step by step. These internals are going to seem rather complicated at this stage, but the reasons for them will become clear later. Of course, as you have just seen, you don't need to be aware of all these details when configuring machines in practice.

The LCFG server daemon continually polls the source files and recompiles them when they change. The logfile for the LCFG server is /var/lcfg/log/server. If you look at this file, you will see the LCFG server noticing the changes you made to the localhost file and recompiling the configuration.

The result of the compilation is an XML file that contains the complete configuration of the machine (or as much of the configuration as we are managing with LCFG). This XML file is /var/lcfg/conf/server/web/profiles/localdomain/localhost/XML/profile.xml. Of course, you can look at this file, but it is not intended for human consumption. The XML file is exported by a standard Apache Web server at the URL http://localhost/profiles/localdomain/localhost/XML/profile.xml.

The LCFG client gets a short notification from the server when the profile changes (it also polls for changes occasionally. in case it misses the notification). When it sees a change in the profile, it downloads the new profile from the above URL. You can see this happening in the client logfile /var/lcfg/log/client:

```
new profile:   http://localhost.localdomain/profiles/
               localdomain/localhost/XML/profile.xml
last modified Fri Jan 25 11:50:07 2008
profile accepted: 93d11c922e99b5d0d26b7c1a8397e4ce
```

The LCFG client then compares the new configuration with the existing configuration. If any parts of the configuration have changed, it calls the corresponding components to implement the change. You can see from the client log that that it has detected a change in the file resources and called the corresponding component:

```
reconfiguring component: file.configure
[OK] file: configure
```

The log for the file component (/var/lcfg/log/file) finally shows the component making the modifications to the motd file:

```
>> configure
configuration changed: /etc/motd
```

The client component normally returns a short acknowledgement to the LCFG server, which keeps a *status page*. This shows which LCFG clients are up-to-date and displays any configuration errors. You can see this using a standard Web browser (in the supplied image, it's Firefox) at http://localhost/cgi/. Clicking on the localhost link will provide details of the individual components.

The command qxprof can be used to inspect the resource values currently in use on any machine. For example, qxprof file will display the resources for the file component. Notice that there are far more resources than those which you explicitly set in the source file. The values for these may have been set as defaults for the component, or they may

have been set in the header files we included in our source file. The -v option can be used with qxprof to display the provenance of all the values.

You might like to spend some time experimenting before reading further. There are a few more example source files (in the same location as the first) which illustrate some other features of the file component:

- example2 shows how the file component can also manage file attributes.
- example3 shows variable substitution.
- example4 shows directory and symbolic link creation.

You might also find it useful to deliberately create some syntax errors in the source file and see how these are reported on the status page. Example 6 contains a deliberate error which is not just a syntax error. You may like to experiment with this and see whether you can locate the problem using the tools above (the answer appears below).

Why Is This So Complicated?

We have installed a lot of software and configuration files. We are running two daemon processes and a Web server. We have written some configuration in a strange language. And the end result is a simple message in one file. Why is this so complicated?

We have only demonstrated with a very simple example, but we are using the full LCFG system. This means that we are now in a position to do much more powerful things with very little additional effort. Here are some examples:

- The LCFG client and server are themselves configured from LCFG resources. This means that we can change their configuration just by adding appropriate resources to the source file.
- By changing one resource, we can run the LCFG client and server on separate machines.
- By simply copying the existing source file, we can configure as many LCFG clients as we like.
- By factoring common resources into shared header files, we can instantly change the configuration of all the LCFG clients (or any subgroup) just by editing the shared header file.
- We can easily add more standard components to manage more of the client configuration.
- We can easily create code to read or write these header files. This could be used to provide a simple Web interface, validate configurations, or generate them automatically.
- We can use an install mechanism to automatically create new machines from scratch. These would be built directly with the specified configuration. We could use this to rebuild replicas of failed machines or to clone machines for a cluster, for example.

2.3 Beyond the File Component

LCFG includes a templating mechanism, which is extremely useful when a few values in a configuration file vary between different machines, but the bulk of the configuration

files remains the same. A common template can be installed on all the machines, and the profiles need only contain the values for the "variable" parts. Since the template itself is the same for all machines, it can be distributed in the same way as the software packages (see section 4.4).

The file component can make use of this template processor automatically. Example 6 uses it to set a specified port number for inbound ssh connections:

```
!file.files             mADD(example)
file.file_example       /etc/ssh/sshd_config
file.type_example       template
file.tmpl_example       /root/sshd_config.tmpl
file.owner_example      root
file.group_example      root
file.mode_example       0600
!file.variables mADD(port)
file.v_port             222
```

As was mentioned earlier, this example contains an error. When you install this configuration, the log from the file component shows that the specified template file does not exist. The file is actually in /root/workshop/part1. Look at this template file and notice the syntax used to substitute the port number from the resources.

Now change the source file to correct the error in the template location and verify that the sshd file is correctly reconfigured.

If we were using this in a real environment, we would probably include all of these resources in a separate header file. This would allow us to add them to any profile with a single statement. Individual machines could override specific values—for example, some machines may want to specify a different port number, which they could do by just overriding the one resource.

Assuming that the sshd configuration file has now changed correctly, try running `ssh -p 222 localhost`. You should find that this does not work, because the ssh daemon has not been restarted. If you restart the daemon manually with `/etc/init.d/sshd reload`, you should be able to log in with the command above.

This illustrates one reason why we might want to use a custom component rather than just using the file component: when the configuration changes, we need to do more than simply change a configuration file. The `sshd` component is customised for managing sshd. It starts and stops the daemon, as well as restarting it whenever the configuration changes:

- Copy the configuration file /root/workshop/part2/example7a to the localhost source file.
- Look at the status page and note that this example has added three new components to the profile—`nsswitch`, `logserver`, and `openssh` (it may take a few seconds for the status page to update). Notice also that the icon for the `openssh` component shows that it has not yet started.
- Look at the sshd configuration file and notice that it has not changed.

- Start the component with om openssh start. When the status page refreshes, it should show that the component has now started (again, there may be a few seconds of delay here).
- Look at the sshd configuration file and notice that it has now been recreated.

Once the component is started (which would normally happen at boot time), it will track configuration changes by rewriting the configuration file and restarting the daemon whenever the openssh resources change. Example 7b contains the necessary openssh resources to set the port number to 222. Install this configuration and you should be able to log in on this port as soon as the reconfiguration occurs (without manually restarting the daemon). You might want to look at the process ID of the ssh daemon before and after the reconfiguration to verify that it really is being restarted. The logfile for the ssh component also records these operations.

The openssh component needs to restart the daemon whenever the configuration changes. This is one reason why you might want to write a custom component (Chapter 6 lists some other reasons).

2.4 Writing Custom Components

LCFG provides a lot of support for writing new components, so it can take a while to become familiar with the framework. Chapter 6 covers this in some depth. But people are often surprised at how little work is required to create a new component from scratch. This section works through a simple example. If you are not interested in understanding how to write your own components at this stage, you can skip this section for now.

```
#!/bin/sh

# The message comes from the command line argument
message=$1

# Save the PID of the daemon so we can find it
echo $$ >/var/lcfg/tmp/chatterd.pid

# Log the fact that we are starting
echo 'date' : chatterd starting >>/var/log/chatterd

# Chatter away—write message to log every 2 seconds
while true ; do
    echo 'date' : $message >>/var/log/chatterd
    sleep 2
done
```

This script is a very simple daemon called chatterd—it writes a message to a logfile every two seconds. The message is supplied on the command line. The process ID is saved in a file so that we can find the appropriate process to kill when we want to stop the daemon. It is included on the demonstration image, and you can try it out manually:

- Run, as root, /usr/lib/chatterd "hello world".
- In another window, watch the logfile with tail -f /var/log/chatterd.
- Stop it (in another window) with kill 'cat /var/lcfg/tmp/chatterd.pid'.

Creating a Component Skeleton

If we wanted to manage chatterd with LCFG, we would probably want to supply the message as an LCFG resource. And we would need to stop and restart the daemon whenever this message was changed. Let's look at what is involved in writing a component to do this.

The command lcfg-skeleton generates "starter versions" of all the files we need for a new component. Try this:

```
$ lcfg-skeleton
Name of component [mycomp] ? chatter
One line description [] ? Example component
Add to CVS (yes/no) [no] ? no
Perl (pl) or Shell (sh) [sh] ? sh
Component author [] ? Joe Smith
Author email [lcfgfc6@localdomain] ? Joe@foo.com
Platforms [Fedora3, ..., Scientific5] ? Scientific5
Include regression test files (yes/no) [yes] ? no
Restart component on RPM update (yes/no) [yes] ? yes
file: ChangeLog
.....
lcfg-chatter not added to cvs
```

This creates a directory called lcfg-chatter with a number of files. We are only interested in three of these for now:

- chatter.def.cin: A skeleton for the schema defining the resources.
- chatter.cin: A skeleton for the component code.
- config.mk: Build-time configuration variables.

Normally, we would create and package the component for production. But we are going to run it directly, so you must *delete the file* test.mk. If you do not do this, the component will be built with test-time pathnames and will not work. Delete this file now.

The LCFG buildtools package includes a script to take the variables in config.mk and substitute them in the .cin file. Type make and then compare the files chatter.cin and chatter to see the substitutions. Type make clean to delete the generated files.

Creating the Schema

First, we need to create a *schema file* (*defaults file*) which defines the resources our component is going to use. If we want to try out the component, we then need to install the schema file and include it in the profile for our machine.

Edit the file chatter.def.cin (*not* chatter.def !) to add these lines:

```
message undefined
@message %string(message): !/^undefined$/
```

The first line defines a new (string) resource with the name message and the default value undefined. The second line is a validation condition which will raise an error if the final value of the resource for any particular machine is left undefined.

Type make install to install the schema file. Notice that this substitutes the variables from config.mk and installs a copy of the schema file in /usr/lib/lcfg/defaults.

We are now ready to add the component resources to our profile. Before you do this, you might want to monitor the LCFG server and client logfiles (in two separate windows):

```
$ tail -f /var/lcfg/log/server
$ tail -f /var/lcfg/log/client
```

Now edit the localhost profile to add the following resources:

```
!profile.components mADD(chatter)
profile.version_chatter 1
chatter.message Hello World
```

The LCFG server will notice the change to the source file and recompile it. You might want to watch the logfiles and check the status page. Once this has compiled, you can inspect the value of the new resource on the LCFG client using qxprof chatter. You might like to try removing the chatter.message resource—notice how the validation condition that we included will flag this as an error.

Now we are ready to write the component code.

The Component Code

The file chatter.cin is a skeleton for the component code. This includes a small amount of code to invoke the framework routines, and three skeleton functions:

- Start() is called when the component is started.
- Stop() is called when the component is stopped.
- Configure() is called when the configuration is changed.

Fill in the body of the functions as follows (take care to copy the quoting correctly):

```
Start()
    Daemon "/usr/lib/chatterd '$LCFG_chatter_message'"
    return

Stop()
    PID=`cat /var/lcfg/tmp/chatterd.pid`
    kill $PID
    return

Configure()
    IsStarted && Stop && Start
    return
```

This makes use of a number of utility functions which are described in Chapter 6. The Start() function starts the chatterd daemon in the background. The Stop() function kills the daemon using the process ID that is stored in the file. The Configure() function simply restarts the daemon to pick up the new value for the message resource. Notice how the value of the LCFG resource is automatically made available as a shell variable.

The component can now be installed just by typing make install. Before starting the component, you might like to monitor the following logfiles (in separate windows again):

```
$ tail -f /var/lcfg/log/chatter
$ tail -f /var/log/chatterd
```

You can now start the component with om chatter start and stop it with om chatter stop. Most important, if you simply change the message resource in the source file, it should propagate through the profile and the component should restart automatically so that the daemon picks up the new message with no manual intervention.

Although we have glossed over a lot of background, you should notice how little code we have had to write to create this custom component (seven lines, apart from the skeleton creation). You might find it worth experimenting with the example a little more before continuing—for instance, you might extend it so that the time interval could be set from a resource rather than being fixed at two seconds.

2.5 Where Next?

We have talked a lot about the LCFG components—how to create new ones, and how they translate the resources into real machine configurations. The real power of LCFG comes from the way that the LCFG server can manipulate the resources to control the configuration of the whole site. The next chapter goes into a lot of detail about the LCFG server and the facilities it provides. Chapter 4 describes some of the standard components that are available.

You can continue to use the VM techniques that we used in this chapter to experiment with all of these new features. For example, you might:

- Create additional virtual machines (either on the same or different physical machines). By changing the LCFG client resource, you could use one LCFG server to configure all of these virtual clients.[1] This would allow you to experiment with a whole simulated site.
- Add additional components to the virtual machines, simply by installing the RPMs off the LCFG Web site. This would allow you to try out all of the LCFG components.
- Write new components to configure an application of your own.

Beyond this, it is necessary to start thinking about how you might use LCFG to manage the overall configuration of your own site. Chapter 5 covers some of these issues.

1. It may be necessary to reconfigure the networking used by the virtual machines to provide the appropriate external access.

3. Writing and Compiling Configurations

The LCFG server compiles the source files describing the configuration of a whole group of machines and generates an individual profile for each one. The term *compile* is rather too grand—the LCFG server and the source language are really quite simple. But it provides several ways of manipulating the resources, letting us think in terms of groups of machines and their relationships, rather than just individual machines.

For example, a very simple mechanism for "including" files lets us factor out groups of resources that define a particular feature. We can then add this feature to a specific machine just by including the appropriate *header file*. There may be a header file which would turn a machine into a Web server or one which would configure its authorisation files as a student lab machine. The source files in the quickstart tutorial include a header file to configure the machine as an LCFG server.

The header files can be included hierarchically, so we can create header files for higher-level concepts such as "student lab machine." These will probably include separate headers for things such as authorisation, software packages, and services. But these different *aspects* are not always completely independent: each header file may contain some resources for several different components. Often, the values for these resources will conflict when the header files are combined. One important feature of the LCFG server is the ability to specify how conflicting resources are handled: one value may take priority, or the final result may be a combination of the two values. For example, two values for the same resource may be combined in a list, or numerically added, depending on the meaning of the resource. This *mutation* is an important feature of LCFG which allows different aspects to be created by different authors without having to resolve every apparent conflict manually.

Another feature of the LCFG server is the ability to create *spanning maps*. These allow the value of a resource to be determined automatically by collating the values of some other resource from a whole set of machines. For example, the IP address of every machine that is configured as a Web server may be automatically added to a particular spanning map. The firewall machine would then be able to use this spanning map to get a list of IP addresses for all the Web servers and automatically use these to configure the HTTP access.

This chapter describes the features of the LCFG server. It starts with a description of the various types of configuration file and explains the configuration language. It finishes with some practical aspects of running an LCFG server, such as monitoring and access control.

3.1 LCFG Configuration Files

All the configuration information for LCFG is stored in plain-text files on the LCFG server. You can back up the configuration of the entire site just by saving this set of files. If you use CVS or some other version-control system (which we recommend), you can roll back the configuration of your entire site to any point in the past. You can automatically generate some aspects of your configuration by writing programs to create or edit these files (a Web interface, perhaps).

In a live production environment, you will probably want to control access to these files and manage the release of changes. Chapter 5 talks about some of these issues.

There are four different types of configuration file: source files, header files, schema files, and package lists.

Source Files

A s*ource file* contains the configuration description for one machine.[1] Every machine being managed by LCFG must have its own source file. In practice, source files usually contain only those configuration parameters that are unique to that particular machine; common parameters are factored out into header files. It is easy to create source files automatically from scripts or GUIs if you have too many to create by hand. A typical source file may look something like this:

```
#include <lcfg/os/redhat71.h>
#include <lcfg/hwbase/dell_optiplex_gx240.h>
#include <inf/sitedefs.h>

dhclient.mac 00:06:5B:BF:87:2E
```

You can use the command `qxprof server.srcpath` to find the location of the source files. The tutorial image uses /var/lcfg/conf/server/source, which is typical.

Header Files

Header files have the extension ".h". They contain common sets of configuration parameters which can be included by the source files or other header files.

Header files contain the same kind of information as the source files—i.e., they define values for LCFG resources. Some header files may contain information about particular hardware, some may contain information about specific site policies, some may contain information about services, etc. The organisation of these header files is very important in managing the overall configuration of a site.

You can use the command `qxprof server.hdrpath` to find the location of the header files. The path may contain multiple directories which are searched in order. You may want to look at some of the header files available on the tutorial image.

1. There are a few cases where source files may not correspond to real physical machines. This is often useful in conjunction with spanning maps. For example, we can create a "dummy" source file for each printer. The information about all the printers can then be collated for use by the print server, or the site inventory.

Schema Files

Schema files have the extension ".def". These are sometimes referred to as *dotdef files* or *default files*. There needs to be one schema file for each component. It serves two purposes:

- It defines the resources the component uses, and possibly some validation or type information for them. This is used by the LCFG server for validating configurations and generating the profile.
- It defines default values for the resources. These values are used in the profile if no explicit value is provided by the source files.[2]

There may be several versions of a schema file for each component, which allows the LCFG server to support clients running different versions of the component. The resource `profile.version_`*`component`* determines the version used by a particular client.

The *schema version* is part of the default filename. There is not always a new version of the schema when the component is updated: the schema needs to change only when the format of the resources is changed in an incompatible way.

The schema files are created by the authors of the corresponding components and installed on the LCFG server. Normally they should not be edited. If you wanted to create a local variant of a schema file, you should probably copy the file and assign it a local schema version.

The default mechanism for building components under Red Hat/Fedora Linux automatically generates an RPM containing the schema file when the component is built. These RPMs have names of the form *name*-defaults-*schema*.

You can use the command `qxprof server.defpath` to find the location of the schema files. The path may contain multiple directories, which are searched in order. You may want to look at some of the files available on the tutorial image.

Package List Files

Package lists specify the software packages to be installed on a machine. Historically, these have been RPMs, and the package list files usually have the extension ".rpms", although the extension ".pkgs" is also accepted. The LCFG server passes this list to the client, where it can be processed by a component. The component updates the installed packages on the machine so that they conform to the supplied list.

The `updaterpms` component processes these lists on a RPM-based system. There is also a component for managing Solaris packages, and it is possible to create a custom component to support any other package format or update technology.

The source file for a machine can specify multiple package list files, as well as directly specifying individual packages. The package list files themselves also support inclusion of further package list files. This means that the structuring of the package lists is important, in the same way that the structuring of the header files is important.

2. Default values provided in the schema file cannot be mutated and are "last resort" values. Working default values (global or site-specific) are usually provided by the header files.

You can use the command qxprof server.pkgpath to find the location of the package lists. The path may contain multiple directories. which are searched in order. Again, you may want to look at some of the files available on the tutorial image.

3.2 The LCFG Language

The source files and header files contain resource values and various directives for manipulating them. Unfortunately, some of the syntax used in these files is not very logical, experimental additions have become permanent, and backwards compatibility with legacy systems has left its mark over the years.

The basic elements, however, are straightforward and powerful. This is the LCFG *source language*:

Resources

All configuration parameters in LCFG are represented by simple key/value pairs known as *resources*. The key consists of a *component* name and an *attribute* name separated by a dot. The value is an arbitrary string which is separated from the key by white space. For example:

```
mailng.relay  postbox@dcs.ed.ac.uk
kdm.greetstring School of Informatics
```

The documentation for the individual components describes the supported attributes. The manual pages are normally available as lcfg-*component.* For example, man lcfg-mailng displays the resources for the mailng component.

The component may also specify some constraints on the allowable values for a resource, and these are validated by the compiler. Some common constraints are often referred to as *types* (e.g., integer), although these are just syntactic constraints rather than any formal type system.

Once a resource value is assigned (either in a source file or any included header file), it is an error to reassign a value to the same resource. Previously assigned values can only be changed using a *mutation* (see below). If no value is supplied for a resource, the default value from the component's schema file is used.

The profile component is a special case. There is no real code for this component; the resources are interpreted as directives to the LCFG compiler. In particular, the resource profile.components lists the components that are to appear in the generated profile. Resources for any components not appearing in this list will be silently ignored. The absolute minimal source file is:

```
profile.components profile
profile.version_profile 2
```

Of course, more components must be declared in order to specify a useful configuration.

The version resource is necessary to specify the schema version of the profile component. This will change if a new profile component is released that has incompatible resources.

Resource Lists ("Tag Lists")

Some resources define a list of items rather than a single value. These are specified in the profile using a structure called a *tag list*. As a data structure, this has some similarity to both the lists and the hashes often found in programming languages; the elements are ordered, but they can also be identified by a named *tag*. The tag can be used to identify individual list elements so that they can be overridden or mutated.

Tag lists are best illustrated by an example, such as this description from the kdm component:

menu: A list of tags for menus to appear on the menu bar
mitem_*tag*: The label for the menu item with the specified tag

Typical corresponding resource declarations might be:

```
kdm.menu file quit saveas
kdm.mitem_file File
kdm.mitem_quit Quit
kdm.mitem_saveas Save As
```

The tags should be unique alphanumeric identifiers.[3] In some cases, the tag names themselves are used by the component; in many cases, they are simply arbitrary identifiers to indicate the resource keys holding the attributes for the list items.

Notice that a source file could override an individual list element by using the tag (the mutation syntax is described later):

```
!kdm.mitem_quit mSET(Exit)
```

Several components make use of multi-level tag lists. For example:

```
fstab.disks hda hdb
fstab.partitions_hda root swap usr
fstab.size_root 100
fstab.size_swap 200
fstab.size_use free
fstab.partitions_hdb home
fstab.size_home free
```

The C Preprocessor

The LCFG compiler passes the source files through the C preprocessor. This allows the familiar syntax to be used for included files, conditionals, macro definitions, and comments. Take, for example, a header file local.h:

```
#include <dell.h>
#define ORGANIZATION ACME Configuration Co
/* Enable this for client debugging */
#undef DEBUG
```

3. That is, unique within a single list. N.B.: It is possible for tag names to include underscore characters, although this can be ambiguous and is not recommended.

This header might be used in a source file as follows (the symbol HOSTNAME is predefined by the compiler to be the name of the current file):

```
#include <local.h>
kdm.greetstring ORGANIZATION host: HOSTNAME
#ifdef DEBUG
client.debug all
#endif
```

Unfortunately, the C preprocessor is designed to process C source code, which does not have the same syntax as LCFG source files. This can lead to problems, because some character strings are mistakenly interpreted by the preprocessor. Comment characters and string quoting are often sources of trouble. The compiler mutation features described below provide some help with quoting awkward cases, but use of the C preprocessor is another design choice that we might make differently next time!

Unlike C, line breaks are significant in LCFG source files, and it is often useful to be able to create macros that generate multiple source lines. The special character ¢ is translated into a newline by the compiler, so multi-line macros can be created, as in the following example:

```
#define BIGDISK \\
fstab.size_root 6000 ¢\\
fstab.size_swap 2000
```

The key sequence Alt-Gr/C can usually be used to produce the ¢ symbol in the source file.

Mutation

Typically, header files provide various levels of defaults for resource values. For example, the LCFG system may ship with a default header file, but you might have a site-specific header file that overrides that value. An individual source file may override this value yet again for one particular machine.

To prevent accidentally overriding resources, the compiler will signal an error if a resource is defined more than once. If you really intend a resource to be overridden, you must use a *mutation*:

```
!fstab.size_root mSET(40M)
```

In this case, a header file has probably defined the default disk partitions for all machines of a particular hardware type, but we want to provide a different partitioning for a specific machine.

The ! prefix indicates that this is a mutation. The mSET() indicates that the provided value is intended to replace any previously defined value. This is often what we want to do, but Figure 3.1 shows some other possibilities. For example, we may want to add an item to a list of values that had been defined elsewhere:

```
fstab.partitions_hda root swap
fstab.size_root 2000
fstab.size_swap 500
```

```
...
!fstab.partitions mADD(var)
!fstab.size_root mSET(1800)
fstab.size_var 200
```

This example produces the following results:

```
fstab.partitions = root swap var
fstab.size_root = 1800
fstab.size_swap = 500
fstab.size_var = 200
```

The ability to *compose* resource values in this way is important if we want lots of people to collaborate on the configuration without conflicting. The standard set of mutations is often sufficient, but it is possible to write custom mutations. For example, we might want to add an extra 40M (numerically) to the size of a partition that is defined elsewhere. Mutations are arbitrary functions (in Perl) that compute a new value for the resource from both the previous value (in $_) and the argument. The file mutate.h is a source of examples. The characters « and » are treated by the compiler as quotation characters and can be used to safely quote the argument even if it contains standard Perl quotation characters. By convention, macros ending in Q expect their arguments to be a quoted string (in Perl syntax), which provides a way of quoting arguments that cause problems with the C preprocessor.

Macro	Description
mSET(A) mSETQ(A)	Override the previous value of the resource with A.
mEXTRA(A) mEXTRAQ(A)	Append the item A to a (space-separated) list.
mADD(A) mADDQ(A)	Append the item A to a (space-separated) list if it is not already present.
mPREPEND(A) mPREPENDQ(A)	Prepend the item A to a (space-separated) list.
mREPLACE(A,B) mREPLACEQ(A,B)	Replace the item A in a (space-separated) list with item B.
mREMOVE(A) mREMOVEQ(A)	Remove the item A from a (space-separated) list.
mCONCAT(A) mCONCATQ(A)	Append the string A to the previous value of the resource.
mPRECONCAT(A) mPRECONCATQ(A)	Prepend the string A to the previous value of the resource.
mSUBST(A,B) mSUBSTQ(A,B)	Replace the substring A with the substring B.
mHOSTIP(L) mHOSTIPQ(L)	Replace any hostname in the (space-separated) list L with the corresponding IP address, by performing a DNS lookup.*

*Care is required when using the HOSTIP functions, because the DNS lookup occurs only at compile time, and subsequent changes to the DNS will not automatically trigger re-evaluation.

Figure 3.1: Standard Mutation Macros

References

It is very important that the components on the LCFG client are independent; generally speaking, components should not rely on the presence of other components, and they should not read one another's resources directly. But it is sometimes useful for the value of one resource to be linked with some resource from another component. The correct way to do this is to use a *reference*. For example, to include the physical location of a machine in the login banner, we would write:

```
kdm.greetstring HOSTNAME (<%inv.location%>)
```

The string <%inv.location%> is replaced by the value of the inv.location resource, which is the physical location from the inventory component. Because this happens on the server, this works even for the inv component, which has no code on the client.

In the above case, the reference is evaluated after all the assignments and mutations. This is known as a *late reference* and is useful because it always evaluates to the final value of the referenced resource, independent of the order. For example, the value of auth.users after the following specifications is john jane.

```
inv.allocated john
auth.users <%inv.allocated%>
!inv.allocated mADD(jane)
```

Sometimes this is not quite what is required. We might want to copy the current value of some other resource immediately, perhaps because we want to perform a mutation on the copy. An *early reference* can be specified using a double percent sign. It will then be evaluated as soon as it occurs. For example, the value of auth.users after the following specifications is simply john (inv.allocated will have the value john jane).

```
inv.allocated john
auth.users <%%inv.allocated%%>
!inv.allocated mADD(jane)
```

C preprocessor macros (e.g., #define) are often used to achieve a similar effect, but the use of references is usually preferable.

Spanning Maps

References enable one resource to refer directly to the value of another resource belonging to the same machine. There is no such mechanism for referencing resources from other machines. This is very similar to programming languages preventing arbitrary access to the local variables of a procedure.

Spanning maps provide a structured and safe way for a profile to access resource values from other machines. The contributing machines will include a component that explicitly *publishes* certain resources to a named *spanning map*. Any machine can then *subscribe* to the spanning map (by name), and the list of collated resource values will be made available as a single resource.

For example, a spanning map can be used to collate the MAC addresses of a set of LCFG clients and make them available to the DHCP server, or to collate the IP addresses of Web servers and make them available to the firewall.

The component author decides which resources will be published to a spanning map and which resources of the subscribing components will receive the collated values (this is described in Section 6.1). As a user of the map, you only need to supply an identifier that provides the link between the publishers and the subscribers. This is usually specified by a resource which is often called cluster. For example, the DHCP clients might declare:

```
dhclient.cluster MYMAP
dhclient.mac 00:08:74:1A:52:7D
```

and the DHCP server might declare:

```
dhcpd.cluster MYMAP
```

The author of the dhclient component has decided that the mac resource will be published to the spanning map named in the cluster resource.

The author of the dhcpd component has decided that it will subscribe to the map named in the cluster resource, that it will import the list of hosts into the host resource, and that it will import their MAC addresses into the corresponding list resources mac_host.

The user has only to supply the map name (MYMAP). All DHCP servers specifying this map name will then serve all the DHCP clients that specify the same map name. By specifying different map names, it is possible to create clusters of machines served by different servers.

Since all spanning map names belong to a single namespace, it is conventional to have map names of the form service/cluster—for example, dhcp/inf1. Notice that clients can be added to or removed from the cluster without changing the server source file.

It is possible for a node to be both a publisher and a subscriber to the same map. In that case, the compiler may require several passes to perform the final evaluation, and this will be detected automatically. Nodes that subscribe to a spanning map will have the publication of their profile deferred until all compilations have been completed. This is necessary to avoid advertising incorrect profiles at intermediate stages of the compilation. This means that it is a good idea to avoid unnecessary spanning map subscriptions.

Package Lists

The LCFG source files may specify a list of *packages* to be installed on a machine. For each package, this includes:

- The package name.
- The version and release.
- An optional *architecture*.
- An optional set of *flags*.

Traditionally, the packages are given as Red Hat RPM specifications which are interpreted by the updaterpms component, but the list can be used to represent packages in any format, provided a suitable component is available to manage them.

The package list could be represented using normal resources, but the LCFG server and client handle the package list as a special case. This provides some useful features and more efficiency. The packages are defined by the profile.packages resource. The value of this resource must be a (space-separated) list of specifications which can have one of three different forms:

name-v-r	The named package is added to the package list. If the specification is preceded by a +, it replaces any previous specification of the same package with a different version/release. If the specification is preceded by a -, any previously defined version of this package is removed from the list.
@*filename*	A list of package specifications in the same format as above (one per line) is read from the named file. The filename should have an extension of ".rpms". By default, an error is generated if the specified file does not exist; appending a ? to the filename will cause missing files to be silently ignored.
tag	The value of the resource profile.packages_*tag* is used as a list of further specifications which are interpreted recursively.

Typically, sets of common packages will be made available in separate files, and individual nodes will select the required sets and perhaps add or subtract a few individual packages. For example:

```
profile.packages dist local
profile.packages_dist @rh71.rpms @rh71updates.rpms
profile.packages_local @local.rpms @private.rpms
.....
!profile.packages mADD(special)
profile.packages_special +foo-1-2 -bar-5-6
```

The first few definitions might occur in a header file; the last two would be specific to an individual node.

The updaterpms component supports a number of flags for controlling various options of the RPM installation (see the manual page for a full list)—for example, preventing the execution of the pre/post install scripts. These flags can be specified by appending them to the package specification with a ::

```
/* Install this package only at boot time */
profile.packages_bootonly foo-3-4:b
```

updaterpms also allows an explicit architecture to be specified if the architecture of the RPM is different from the default (i386). For example:

```
profile.packages_mp3 notlame-3.92-*/i686
```

Contexts

The general philosophy of LCFG is that configurations should be computed completely on the LCFG server. This allows us to identify simple errors before the configurations are deployed. It also means that inter-machine dependencies, such as spanning maps, can be computed without any network communication between machines.

However, there are a few cases where it would be unreasonable (or impossible) to compile a new configuration on the LCFG server every time. For example:

- We might want to make a student laboratory machine available to remote users outside of opening hours (so the authorised user list might change at different times of the day).
- We might want the set of packages to be included at initial install time to be slightly different from the packages to be loaded when the client is fully installed.
- The mail relay on a laptop may need to be different according to the ISP that is being used.

All of these cases could be handled by custom code in the appropriate components, but there are some advantages in having a uniform way of supporting two common situations:

- We want to switch easily between two (or more) values for a resource (although the set of values can be predetermined by the LCFG server)—for example, two different user groups for login authorisation.
- We want to override a value for some resource using data that is only available to the client—for example, the mail relay.

LCFG provides a generic *context* mechanism which can deal with both of these cases. The LCFG client maintains an arbitrary set of *context variables* which can be set to arbitrary values, using the context command. For example:

```
$ context
dock=home
$ context stuff=foo
$ context
stuff=foo
dock=home
$ context stuff=
$ context
dock=home
```

The source file can specify several different values for a resource, to be used in different contexts. For example:

```
mailng.relay mailhub.ed.ac.uk
mailng.relay[scheme=home] mail.myisp.com
```

In this example, when the context variable scheme has the value home, the mail component will use mail.myisp.com as the relay; in all other cases, it will use mailhub.ed.ac.uk. (Contexts are persistent, even across reboots.)

The conditional context expressions must appear in square braces immediately after the resource attribute. They can include:

var	True if the named context variable is set (non-null)
var=value	True if the context variable has the specified value
var!=value	True if the context variable does not have the specified value
expr1&expr2	Logical AND
expr1|expr2	Logical OR
!expr	Logical NOT
(expr)	Braces

In addition to switching among a number of predefined values, it is also possible to set the value of a resource locally (on the client). This might be necessary, for example, if we wanted to obtain the mail relay from DHCP. The details of this are not explained here, but the LCFG wiki shows some examples of the technique being used for debugging.

The context processing is implemented in the LCFG client. When the context changes, the components just see a normal configuration change—they don't need to be aware of whether this is caused by a source configuration change or a context change. The context changes can be initiated manually, from cron, or by any other program (simply by using the context command).

A Warning

The context mechanism does subvert many of the advantages of LCFG, and it should be used sparingly. If you are thinking about using a context, the following points are worth noting.

- The LCFG server cannot perform validation on context-dependent resources.
- Features such as mutation and spanning maps are not available.
- Some resources are evaluated on the LCFG server rather than the client (e.g., the profile component, the inv component). It makes no sense to attach context expressions to these resources.

It is an error to specify a context-specific resource without a context-free specification of the same resource. If there are multiple context-specific resources that match, the most

recently set context takes precedence. Conditionals that depend on multiple context variables require careful construction to ensure that they are always disjoint, and this is best avoided.

Since the profile resources are interpreted by the compiler, context specifications cannot be attached to the profile.packages resources. However, as a special case, context specifications can be appended to any package specification, whether it appears inside an rpmcfg file or explicitly in a source file. This is often used to prevent packages being installed during initial node installation. For example:

```
/* Do not install big packages at install time */
profile.packages mADD(bigstuff)
profile.packages_bigstuff bigpack-3-4[!install]
```

3.3 The LCFG Server

The program mkxprof is the LCFG compiler. This takes configuration descriptions in the source language and turns them into XML profiles (one per machine). mkxprof can be run manually to compile a single profile:

```
$ mkxprof host035
** conflicting package specifications: p
**   p-5-6: (/TEST/src/host035:7)
**   p-8-9: (/TEST/packages/packages035.rpms:4)
** unrecognised package spec: tag2 (/TEST/src/host035:6)
```

After a successful compilation, an XML profile will be generated in the appropriate directory (use mkxprof -V to see the default directories). This will usually be published by the Web server to make the profile available to the client.

mkxprof can also run in daemon mode. In this case, it maintains a database of source file dependencies and continually polls for changes. When a file is changed, it recompiles all of the dependent profiles. It can also generate HTML status pages for each profile to display error messages, rather than requiring them to be retrieved from the LCFG server logfile.

Manual compilation is useful for simple testing, but in practice, you probably want to supply a lot of options to mkxprof. The LCFG server component manages the mkxprof daemon and allows you to supply all of the options as LCFG resources. In production, mkxprof is almost always run in this way.

When a profile changes, the LCFG server sends a simple UDP notification to the client, without waiting for an acknowledgement. The client polls the LCFG server at regular intervals in case it misses a notification. When the client sees that a new profile is available, it fetches the XML using normal HTTP from the LCFG server. The XML is parsed and the resources are stored in a local database.

The client component attempts to optimise profile fetches and parsing by only performing these operations when it believes that they are necessary due to a change. The install method of the client component can be used to force a new copy of the profile to be fetched from the LCFG server and re-parsed. The install method can also be provided

with an explicit URL as an argument; this can be used to force the client to fetch the profile from a different LCFG server.

Any components whose resources have changed are called to perform a reconfiguration. Exactly how and when the component decides to implement the reconfiguration depends on the particular component. For example, some things can be changed immediately, other things may need to wait until the user has logged out or until the node is rebooted.

The current resource values being used by a client can be queried using qxprof. If the client is running the logserver component (see Section 4.2), the resources can also be inspected remotely via the links on the HTML status page.

Authorisation and Access Control

Different sites will manage the LCFG source files in different ways, and they will use different mechanisms for controlling access. The contents of the LCFG profiles should be considered public, though; any truly sensitive information should be encrypted at the application level, because the profile is plainly visible on both the LCFG server and the LCFG client.

However, the LCFG server does provide a simple mechanism for automatically generating per-profile .htaccess files to provide some degree of control. These can be created using the resources:

```
profile.auth          http ssl
profile.file_http     .htaccess
profile.file_ssl      sslaccess
```

and can be included in the Apache configuration with:

```
<VirtualHost *>
  AccessFileName .htaccess
</VirtualHost>

<VirtualHost *:443>
  SSLCertificateFile /usr/share/ssl/certs/mycert.pem
  SSLCACertificatePath /usr/share/ssl/certs
  SSLEngine on
  AccessFileName .sslaccess
</VirtualHost>
```

An access control string specifying permitted IP address ranges can be given for each access control file:

```
profile.acl_http      <%profile.node%>.<%profile.domain%>
profile.acl_ssl       129.215
```

Basic authorisation directives can also be specified. These apply in addition to any access control; if the access control directives are not present or if they deny access, the username and password can be used to gain access:

```
profile.passwd       foobar
profile.pwf_http     auto
```

The profile.passwd resource causes the LCFG server to automatically create an Apache-compatible DB password file and make an entry for the fully qualified hostname with the given password. The second resource permits access to any client using the HTTP protocol and supplying the given password (with the FQDN as username).

The LCFG client will cache any password that is defined in a profile and use this password when making future requests. For example, a laptop may be initially installed on the local network, where the access control permits the profile to be downloaded freely. This profile contains the initial password, which is then used for subsequent requests when the laptop is operating remotely and authorisation is required.

The LCFG server provides a mechanism for linking arbitrary directories to the exported Web tree. By default, this is used to publish the directories holding the status CGI scripts, the help files, and the icons:

```
server.linksdirs     cgi help icons
server.src_cg...
server.dst_cgi       ...
...
```

The access control for these directories can also be set from the resources. For example:

```
server.auth_cgi      hhtp
server.file_http     .htaccess
server.acl_http      129.215
server.pwd_http      auto
```

The Status Display

The LCFG server can provide HTML pages showing basic status information for each client. This is not a substitute for a full monitoring system, but it does provide a good overview of the configuration state of the site. Normally these pages are generated on the fly by CGI scripts that read the LCFG server status (see figures 3.2 and 3.3, p. 32). They show information from three sources:

- Static information obtained by the LCFG server when compiling the profiles. This includes basic inventory information and any compilation errors.
- Information returned by the client in simple UDP acknowledgement packets. This includes some simple monitoring information from the running client (e.g., the boot time) and basic status information for each component on the node (is it running? has it generated any errors? etc.).
- The node itself may be running a logserver component. This is a small Web server which makes the LCFG logs, and other detailed information available directly from the client itself via HTTP. If this component is running, the status page will provide links to the appropriate URLs.

The status page is normally available at http://lcfg-server/cgi/index.cgi.

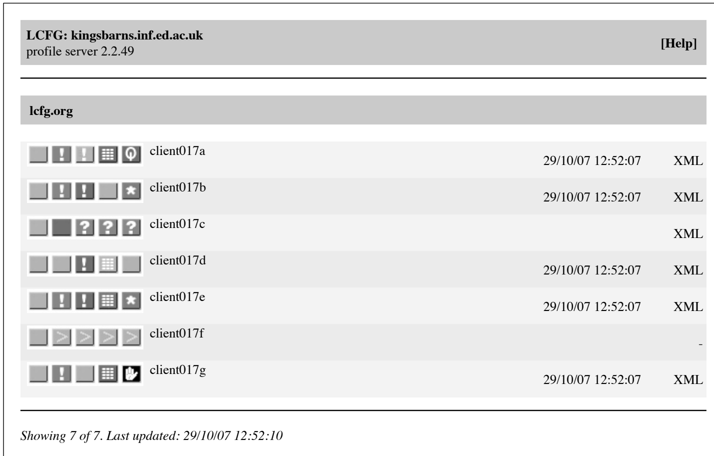


Figure 3.2: Summary Page

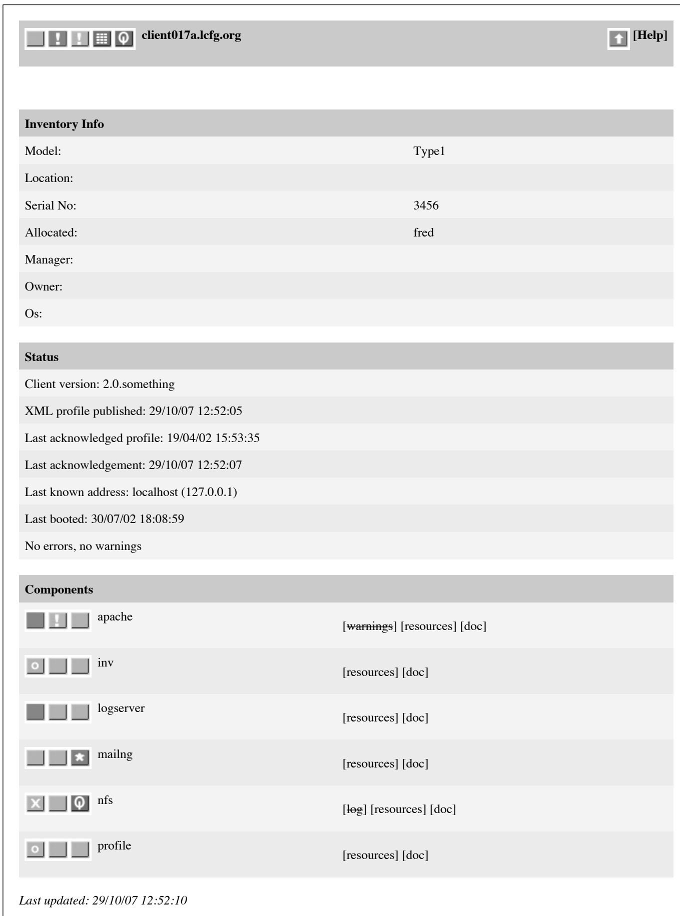


Figure 3.3: Individual Client Display

A few points worth noting about the status display:

- The client normally sends acknowledgements when polling for a new profile or whenever an event change (e.g., an error) occurs. A throttle algorithm prevents clients from sending rapid acknowledgement streams, and this introduces a slight delay in notification.
- The main display is only updated at the end of a server pass. The frequency depends on the LCFG server resources, but there may be a significant delay (20 minutes, for example) if the LCFG server is recompiling a large number of profiles. The main display may also be out of sync with the individual client displays during this time.
- Clients will be marked as "late" if no acknowledgement has been received within the latency time. This time is the maximum time that would normally be expected between client acknowledgements. It is based on the sum of the poll times of the client and server components.
- Error and warning conditions can be reset by calling the `Reset()` method of the offending components or by rebooting.
- If client nodes have an `inv` component in the profile, the LCFG server will publish the inventory fields listed in the `inv.display` resource on the status page.

4. LCFG Components

LCFG *components* are the scripts that are responsible for configuring the various applications on the client to match the resources specified in the profile. In Chapter 2, we saw how the file component can be used to manage arbitrary configuration files. But there are a number of reasons why we might want to use a custom component instead:

- Some action needs to take place when the resources change or when the components start or stop. Typically, this involves managing a daemon such as openssh or sendmail.
- The changes to a configuration file need to be computed from the resources, but this is more complex than a simple substitution. For example, the xfree component will automatically probe for the monitor type if the resource value is specified as "auto."
- A resource needs to be validated in a specific format at compile time: for example, to ensure that a valid URL is provided before the profile reaches the machine.
- A spanning map is required to collate the resources from several machines and make them available to some other machine. For example, the static IP addresses of DHCP clients are collected and made available to the configuration of the DHCP server.
- A number of resources are involved, and it is simply clearer and more modular to use a separate component (e.g., nsswitch).
- A small number of components have special roles and are automatically called from other parts of the system. For example, the boot component is called at system boot time, and the updaterpms component is used to update software packages.

In this chapter, we will look at some of the features of the *component framework* that is common to all the components. Then we will look at a few important components in more detail. (Chapter 6 explains how to write your own component.)

4.1 The Anatomy of a Component

Components are just simple scripts. But they are written using a standard framework: they all accept a standard set of *methods* as arguments. All of the methods can be invoked from the command line by using the om command, but they are often called automatically by other parts of the system. Most methods are protected by locks to prevent multiple simultaneous invocations.

Common Methods

- configure: This is the most important method. It is called by the LCFG client whenever the component resources are changed. The component updates the configuration files to reflect the new resource values, and it notifies any associated daemons. Usually this happens immediately, but sometimes it is more appropriate for the component to delay the reconfiguration: for example, until the user has logged out, the gdm component will defer updates that involve restarting the window system.
- start: This is called at boot time to "start" a component. If the component manages a daemon, for example, the start method will physically start the daemon. Even if the component has no specific operations to perform at start time, the method is still important, because the framework will not issue configure calls to components that are not started. This method calls configure automatically, to ensure that any configuration files generated from the resources are up-to-date before starting any daemons.
- restart: This operation is the same as start, except that the component is first stopped if it is already running.
- stop: This method is called to stop the component at system shutdown. The component stops any running daemons.
- run: This method is typically called from cron, or manually, to perform some ad hoc operation.

Less Common Methods

- logrotate: This method is conventionally called by a logrotate script to notify any daemons that they should release logfiles.
- status: This method prints the current state of the resources being used by the component. If an update is pending, they may not be the same as the resources currently specified in the profile. Some components may use this method to make other status information available, when a component-specific option is given. There is no lock on this method.
- monitor: This method is used to request that the component report monitoring information. The first argument is a tag identifying the type of monitoring information requested. This method is rarely used and is ignored unless the component has been configured.
- reset: This method clears the error and warning files that are used by the status display to determine the icon indicating the component status.
- unlock: This method forces removal of any locks.

Suspend and resume methods also exist for processing APM suspend/resume events, although these are not currently in production use and may not be reliable.

Some components may define additional, custom methods, although this is discouraged, and the use of custom options to standard methods (such as run) is preferred.

Om

Components are simple scripts. It is possible to call them directly, just by giving the method as an argument:

```
$ /usr/lib/lcfg/components/logserver stop
```

But the command om is the preferred way of calling components:

```
$ om logserver stop
```

This provides access control for non-root users, remote execution, a standard environment, and transparency in the location of the scripts. The full syntax of the command is:

$ om [*hostname*] *component* *method* [*options*]

If the *hostname* is present, ssh is used to call om on the remote host.

Access control for non-root users is specified using the following (per-component) resources:

om_methods: specifies the allowed methods.

om_authorization: specifies the Perl module to be used for performing the authorisation.

om_user: specifies the username under which the component is to be run.

om_acl_method: specifies the authorisation token for the method method. The exact meaning of this token depends on the specified authorisation module.

The default authorisation module is LCFG::Authorize, which allows the permissions to be specified as resources in the LCFG source file. For example:

```
mailng.om_acl_start om/mailctl
mailng.om_acl_stop om/mailctl

om.groups admin
om.users_admin john jane
om.caps_admin om/mailctl
```

Larger sites may want to replace LCFG::Authorize with a module that integrates with their own authorisation system—we use a module that interfaces with an LDAP-based capability system.

Method Options

The standard component framework accepts a number of generic options which can be specified following the method name:

-d: dummy. The component actions are printed but not executed. This is not always easy to implement, and the effectiveness of this option depends on the individual component.

-D: debug. Print debugging information.

-n: no strict. Certain warning and error messages are suppressed. For example, trying to stop a component that is not started will normally generate a warning message. If this option is used, the warning is not generated.

-q: quiet. No messages are printed.

-t timeout: set lock timeout. Normally, if a component is already executing, calls to most methods will block until the existing instance terminates and releases the lock. This option specifies a timeout so that the current call will terminate after timeout seconds if the lock cannot be obtained. Certain method calls do not lock (see the list above), and locks can be broken using the unlock method.

-v: verbose. Additional messages are printed. Holding down the shift key (on some platforms) when a component method starts executing will also enable this option. This is useful at boot time to enable more verbose logging on certain components.

Components sometimes define additional component- and method-specific options. If present, these must be separated from the generic options by --. For example:

```
$ om updaterpms run -- -t
```

4.2 Some Common Components

LCFG is completely modular, and different machines will usually be running different sets of components. But a few components are closely tied to the operation of LCFG itself, and they will usually be present.

The manual pages for a component can be viewed with man lcfg-*component_name*.

The Profile Component

The profile component is not a “real” component, in the sense that there is no code on the client. The LCFG server uses the profile resources as directives to control the compilation of the client profile, including access control, the format of the profile, and the list of other components to be included in the profile. This is the only component that is mandatory in every profile.

The Client Component

The client component manages the rdxprof daemon. It watches for changes to the published profile, downloads new copies, parses the profile, and calls the configure method for any components whose resources have changed. This is necessary for any machine that is going to be managed by LCFG.

The Boot Component

LCFG components can be started at boot time just by including them as part of the default system init process (the demonstration disk does this). However, the boot component is a complete replacement for the native init.d mechanism, which allows the servic-

es and their order to be determined from the LCFG resources rather than fixed files. It also allows services to be started or stopped dynamically when the configuration changes.

The component can manage a mixture of LCFG components and traditional System V init scripts. For example, you could use the following resources to add the System V init script ypbind and the LCFG component mailng to the list of services started at boot time:

```
!boot.services mADD(rc_ypbind)
!boot.services mADD(lcfg_mailng)
```

Note the use of the prefixes rc_ and lcfg_ to distinguish the two different types of service.

The boot component can also arrange to call component run methods from a single cron job (normally, nightly).

The File Component

You have seen the file component in action in Chapter 2. This is a general-purpose component which can be used to easily create and customize configuration files, directories, or links. This allows you to configure simple applications without the need to write a special component.

Resources are used to specify a template file and values to be substituted into the template. The template is normally installed site-wide, and the values from the profile are used to configure the file and customize it on a per-machine basis.

For example, we could distribute a template (containing variable references) for the php.ini configuration file (call it php.ini.tmpl):

```
...
engine = <%v_phpenable%>
...
```

We could then configure the file component to create the php.ini file from this template:

```
!file.components     mADD(file)
!file.files          mADD(php)
file.type_php template
file.file_php        /etc/php.ini
file.tmpl_php /etc/php.ini.tmpl
```

and set the default values for the variables:

```
!file.variables mADD(phpenable)
file.v_phpenable     On
```

Individual machine configurations can now control the php engine simply by setting the value of this variable in their source files.

Very small templates can even be included in-line in the resources, which avoids the need for a template. For example, the bluez.pin file needs to contain only a PIN number:

```
!file.components        mADD(file)
!file.files             mADD(bluez)
file.type_bluez         literal
file.file_php           /etc/bluez.pin
file.tmpl_php <%v_bluezpin%>
!file.variables mADD(bluezpin)
file.v_bluezpin         1234
```

Other uses of the file component include the creation of user home directories, arbitrary links, and the ability to control file attributes.

If several different applications are being configured, it is often convenient to assign each application a separate schema file so that it may use its own variable namespace. The file component supports such *managed components*, still without the need for any special component code.

The Inventory Component

The inventory "component" is really a pair of "pseudo-components."[1] This is used to collate inventory information from all of the profiles and make it available in a single file. This is not an important component, but it illustrates an unusual application of spanning maps.

The inv component can be included in normal profiles and be used to define basic inventory information for the machine (see the manual page for lcfg-inv for details of the available fields). For example:

```
!profile.components mADD(inv)
inv.model Dell Optiplex
inv.allocated fred user
inv.manager the boss
inv.location myroom
```

This information is published to a spanning map. The inventory component can be included in the source file of a dummy machine to import the spanning map:

```
profile.components profile inventory
profile.version_profile 2
profile.version_inventory 1
profile.format XMLInventory
profile.ng_statusdisplay false
```

The profile for the dummy machine now includes inventory information from all of the subscribing machines.

This example also illustrates the use of a special *profile* format. The "profile" for the dummy machine is formatted as a simple XML inventory:

```
<node name="red">
```

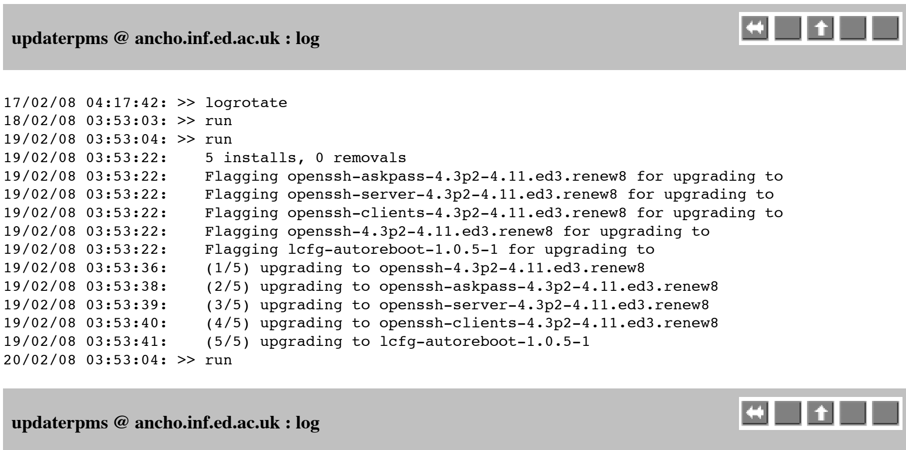
1. The term "pseudo-component" is usually used to refer to a component which has no corresponding code on the client. The resources for a pseudo-component are read and processed by some other application—for example, the server itself, or a different component (via a spanning map).

```
        <model>Dell Optiplex</model>
        <allocated>fred user</allocated>
        <manager>the boss</manager>
        <location>myroom</location>
        ...
    </node>
    <node name="blue">
        ...
    </node>
    ...
```

Of course, this is not a real profile—it can't be used to configure a machine. It would be read by some application to process or search the inventory.

The Logserver Component

The logserver component makes LCFG log information available via HTTP. If this is running, the LCFG server will display links on the status page that provide direct access to the logfiles for all of the components.



Notice that there is no access control on the published logfiles, and the logserver resources will normally be used to restrict access to any file containing sensitive data.

Figure 4.1: Logserver Display

4.3 More Components

In addition to the components we have already seen, here are some others available from the Web site:

> lcfg-afs, lcfg-alias, lcfg-amd, lcfg-apache, lcfg-arpwatch, lcfg-auth, lcfg-client, lcfg-condor, lcfg-cosign, lcfg-cron, lcfg-cvs, lcfg-cyrussasl, lcfg-dhcpd, lcfg-dns, lcfg-etcservices, lcfg-example, lcfg-file, lcfg-fstab, lcfg-gdm, lcfg-gridengine,

lcfg-grub, lcfg-inventory, lcfg-iptables, lcfg-kernel, lcfg-kx509, lcfg-mailcap, lcfg-mailng, lcfg-matlab, lcfg-mozilla, lcfg-mysql, lcfg-network, lcfg-nfs, lcfg-nsswitch, lcfg-ntp, lcfg-openldap, lcfg-openssh, lcfg-openvpn, lcfg-pam, lcfg-perlex, lcfg-postgresql, lcfg-ramdisk, lcfg-rmirror, lcfg-rsync, lcfg-samba, lcfg-server, lcfg-snmp, lcfg-subversion, lcfg-syslog, lcfg-tcpwrappers, lcfg-updaterpms, lcfg-webdav lcfg-xfree,lcfg-xinetd

Many of these components will be quite small, and the production quality will vary, but this list illustrates the range of services supported. These also make a good starting point for writing your own components.

4.4 Updating Software

The LCFG components we have seen so far have all been concerned with making changes to configuration files and controlling any associated daemons. We haven't said anything about how these configuration files, daemons, and other software get installed on the machine and updated.

LCFG doesn't mandate any particular technology for installing software packages. What it does do is give you a way of managing the list of packages that should be installed. You can then create a component using your favourite technology to handle the package installation and removal. Our Linux systems use the updaterpms component, which relies on the updaterpms program to manage RPM packages. Other components have been written for Solaris and Mac OS X packages.

The software update component adds and removes packages from the machine to make it conform to the list specified in the profile. It also notifies the LCFG client (by touching the file /etc/LCFG-RELEASE) when an update has been successful. This allows the status page to display a warning for those hosts that have not had successful updates for a specified length of time. The contents of the LCFG-RELEASE file (installed from a package) may also be used to give an identity to the overall "release" of the installed software. This too can be checked by the LCFG server and flagged if it is not as expected.

The following sections explain the handling of the package list (which is independent of the update technology) and then describe the components available for managing RPM packages.

The Package List

The client component downloads the list of packages created by the LCFG server (see section 3.3) and stores it in /var/lcfg/conf/profile/rpmcfg/nodename. This file is updated every time a new profile is received with changes to the package specifications.

A software update component (such as updaterpms) needs to implement a run method. When this is called, the component should remove and add packages as necessary to synchronise the installed packages with the package list. There are several possibilities for configuring exactly when the run method is called:

- The client.runupdate resource can be set to initiate a software update immediately whenever the list changes. But this is likely to be disruptive for the user, so one of the following methods is often used.

- The update component is added to the boot.run resource so that it is called whenever the boot component runs. Normally this happens once nightly—from cron, as specified by the cron.run_boot resource. Of course, the run method could also be called directly from cron using om.
- Sometimes there is a need for more control over the timing of the updates (e.g., on a laptop). In this case, the updaterpms component can be run manually. Non-root users can be permitted to do this by setting the updaterpms .om_acl_run resource to a capability for the user, or simply to <console> (for any user at the console).

Some packages (e.g., the kernel) should not be updated on a running system. updaterpms provides a flag to indicate that such packages should only be installed at boot time.

The package list contains one package specification per line, in the following format:

```
name-version-release
```

The version and the release may contain wildcard expressions, which are interpreted by the update program to mean "the latest available." The supported syntax of these expressions, and their evaluation, depends on the update program. The manual page for updaterpms describes the format accepted by that program.

The use of wildcard versions is very convenient during development—new versions of packages can be easily installed without changes to the profile. This might not be such a good idea for production installations, though, because it is no longer possible to tell, just from the profile, exactly what software is installed on each machine.

If the required architecture is different from the default, the package specification may be followed by an optional architecture. This may be followed by a : and a number of single-character flags. The meaning of these flags depends on the update program being used.

The package list is designed to be passed through the C preprocessor (cpp) and contains several cpp directives:

- #include can be used to include local RPM lists.
- #ifdef is used to allow different sets of RPMs to be selected.
- #pragma LCFG derive gives the location(s) in the LCFG sources where the packages were specified (if known). This is useful for debugging.
- #pragma LCFG context gives the context in which the following package was specified.

Normal software update components can ignore the significance of the conditional commands and context pragmas. These are used by the rpmcache component, which needs to cache all the packages, regardless of their context.

Updating RPMs

The updaterpms program manages packages for RPM-based systems. It compares the installed RPMs with the RPMs specified in the package list. It then installs/updates/deletes RPMs to make the set of installed packages correspond to the required list.

The updaterpms.rpmpath resource specifies a (colon-separated) list of *repository* locations. Each location may be a pathname or the URL of an HTTP-exported directory. These are searched (in order) for the specified RPMs.

Every RPM in the repository must also have a corresponding file with the same name, prefixed by a dot. This file contains meta-information for the package and is used by updaterpms to avoid the overhead of reading the entire RPM file to extract this information. The program genhdfile is used to generate these files:

```
$ genhdfile mpdist-3.5.2-2.i386.rpm
$ ls .mpdist-3.5.2-2.i386.rpm
.mpdist-3.5.2-2.i386.rpm
```

Repositories that are exported via HTTP must also contain a file called rpmlist, which simply lists the RPMs in the repository, one per line. This could be generated by the commands:

```
$ cd repository
$ ls .rpm >rpmlist
```

If the rpmlist or hd files are missing or out of date, updaterpms will generate errors or install incorrect versions. It is usual to have a script that updates these files automatically.

The RPM Cache Component

The rpmcache component allows a cache of RPMs to be maintained on the local disk. This is useful in several cases:

- The updaterpms component installs RPMs as they are downloaded. Especially if the network connection to the repository is unreliable, it may be helpful to ensure that all the necessary RPMs are available on the local disk before commencing the update.
- If a node (e.g., a laptop) is liable to be disconnected from the network, a local cache of RPMs can be used to reinstall or check the installation of individual packages without being connected to the network.
- A local cache of RPMs can be re-exported as a repository to other LCFG clients.

Typically, the RPM cache component is configured to fetch the RPMs from the remote repositories and to trigger updaterpms when it has finished. updaterpms is configured to use the local cache as its repository.

5. Managing a Site with LCFG

By now, you should have a good idea of the mechanics of LCFG. But managing a real site involves a lot more—the manual procedures and the "mindset" behind the use of the tool are equally important. The SAGE *System Configuration* booklet provides some general discussion of these topics. In this chapter, we will look specifically at how LCFG can help.

5.1 How Much to Automate?

There is always a trade-off between the effort involved to perform a task manually and the effort involved in automating it. Automation entails an initial development cost and it requires certain specialist skills, but the resulting system should be less effort to manage and more reliable. In general, the larger the site, the more viable automation becomes.

In theory, more automation should make a system:

- More reliable and "correct"—i.e., it behaves according to the requirements.
- More efficient—i.e., it takes less effort to manage.
- More flexible—i.e., it is easier to accommodate changes in the requirements.

But the automation has to be managed well to reap these benefits. It is easy to imagine a site where people spend most of their time writing ad hoc "time-saving" scripts. If these scripts are fragile and poorly understood, they can easily contribute to a very unreliable and inflexible system.

The traditional, bottom-up approach to configuration starts with scripts to solve small problems and attempts to evolve into a solution for the whole site. This can produce poorly structured solutions which are not well suited to automating the higher levels—for example, the co-ordination between Web servers and the corresponding holes in the firewall.

LCFG provides a framework for developing a configuration solution. This allows a site to start with the simple use of the `file` component to manage one or two configuration files. From there, it can grow, by gradually adding components, into a fully prescriptive system. The framework helps to organise the configuration process and ensure that it remains manageable as it expands. An installation that is managed completely by LCFG usually provides a solid foundation for automating higher-level aspects of the system.

Most people find that there is a significant cost to getting started with LCFG. Once they are familiar with the system, automation tends to grow fairly quickly and in a controlled

way. This often ends up being a fully prescriptive system. This has a lot of benefits, but it is not essential; even in a mature system, there will be cases where the cost of writing a component for a temporary application will not be worthwhile.

As we noted in the introduction, one thing is very important: a clear separation between those parts of the system that are managed by LCFG and those that are not. Having decided to automate some subsystem, LCFG will expect to have control over it, and it needs to be trusted. Conflicts between manual intervention and one or more configuration tools are a common source of errors.

5.2 Who Manages What?

Traditionally, individual system administrators tended to have complete control over "their" servers; one person will have been responsible for all aspects of a machine, including installation, software updates, configuration changes, etc.

In a large, modern installation, this is not feasible. Many different people need to have a say in the configuration of a machine. To give only a few examples:

- The security specialist may want to control various authorisation and firewall settings.
- The end user may want to specify the versions of application packages to be loaded.
- The networking specialist may want to configure the DNS.
- The platform specialist may want to update the OS software.
- The finance department may want to update an application client.

These are too varied and too complex for a single administrator to understand fully. More importantly, these *aspects* often have implications for other machines on the network (e.g., when updating the finance software, all the clients and the servers must be updated at the same time). This synchronisation cannot be allowed to depend on manual communication between the administrators for all of the machines involved.

Large installations tend to have a range of specialists. The configuration of any one machine is made up by *composing* their individual contributions. LCFG recognises this and provides a mechanism so that a number of people can collaborate on the configuration of a machine with a minimum of conflict.

For example, the file /etc/hosts.allow typically specifies which services are provided to which remote hosts. Different entries in that file may well be the responsibility of different people (ssh, mail, snmp, etc.). Traditionally, there might be some conflict over the ownership of that file. Some tools, such as Cfengine, are designed to edit parts of the file without affecting other parts, but this is very hard to do reliably.

LCFG avoids this problem completely. Different people can specify access resources for different services (usually in different header files). The LCFG compiler composes these, resolving any conflicts, and the component regenerates the complete hosts.allow file.

If there are conflicts in the values specified by different people, the LCFG server can often resolve these automatically—perhaps by giving one aspect priority, or perhaps by

creating some composition of the conflicting values. If any unresolved conflicts remain, these will be reported at compile time rather than misconfiguring the machine.

LCFG's mutation functions are central to this composition process. Rather than having to specify a complete list of services, for example, it is normal to use `mADD()` to "add my service to the ones already there." Or "add 50 MB to the size already specified for this partition."

Header File Organisation

The organisation and structure of the header files are important issues for an LCFG site. In general, the header files are created to represent different aspects of the configuration—perhaps a physical aspect such as the machine model or a logical aspect such as a "Web server." These headers are created by the various specialists. The more general system administrators can then select the appropriate ones to configure a particular machine.

Typically, the headers will be structured into some kind of inclusion hierarchy. There are general lcfg-level headers, which contain generic defaults. These will be included in site-level headers, which override some values to make them site-specific. These may be further included in department-specific headers, and so on.

A mature LCFG site will have a number of header files, so the structure of the hierarchy is important. In particular, the mutation functions are often sensitive to the inclusion order of the header files. It is also common to use preprocessor commands to detect particular combinations of headers and make specific adjustments to the configuration. This can be fragile if it is not used with care.

Finally, there is no access control on individual LCFG resources—access control can only be implemented at the file level. But source files may contain arbitrary resources, and even preprocessor commands, so that level of access control is ineffective. The LCFG source files are simple plain-text files, though. It is common to generate some included files, perhaps from a utility or Web form which provides control over a restricted number of resources and their values.

5.3 Managing Change

Some sites demand a very stable configuration. For others, however, the hardware, the software, and the requirements change almost continually. LCFG is capable of supporting a high rate of configuration change. It is not uncommon to see a large LCFG server almost continuously busy tracking changes to various source files.

Especially when there is a wide diversity of configurations, though, high rates of change make it very hard to test new configurations properly. Of course, errors can usually be fixed quickly when they are detected, but it is usually preferable to "stage" the changes, so that production systems are more stable. LCFG makes this very easy. For example, we use a scheme similar to the following:

- During development, new changes are deployed only onto an LCFG "development" server. The developer's test machines are configured from this server.

- At various intervals, the configuration files from the development server[1] are transferred to an LCFG "test" server. This is used to configure a small set of machines that are typical of the production environment. The new configuration release is tested for a few days on these machines.
- Usually weekly, the production LCFG server is synchronised with the test server to release the new configurations onto the production machines.

Of course, it is not possible to test all aspects of the configurations, and the test machines represent only a sample of the production machines, so errors do still occur in production. But the LCFG sources are maintained in CVS, and it is usually simple to roll back the configuration of the entire site to any point in the past.

Some configuration changes clearly need to bypass this mechanism. If the network is being reconfigured or a server is being moved, that aspect of the production configurations needs to be changed at the appropriate time. A small number of header files are shared directly among the three LCFG servers ("development," "test," and "release").

LCFG does support a mechanism to allow configuration changes to be manually vetted before they are accepted on the client. This may be appropriate for a few highly critical servers which need to be very secure and/or very reliable (e.g., the Kerberos master).

Some configuration tools allow more manual intervention in the configuration deployment. This is strongly discouraged with LCFG, because it is often managing the configuration of relationships between machines. If a server is reconfigured, the client must be reconfigured to match. Delaying the reconfiguration of the client will simply break the service.

5.4 Installing from Bare Metal

The ability to (re)install machines easily from bare metal[2] is very important. This is what lets us replace failed hardware or duplicate an existing machine to increase the size of a cluster, for example. Various "cookie-cutter" mechanisms have been available for a long time—these usually duplicate an existing disk image to create a clone.

However, cloning does not always work well. Duplicating machines to increase the size of a cluster requires us to store and maintain too many different images. More subtly, we often want to clone a logical part of the configuration rather than the whole thing—for example, we might want to recreate the functionality of an existing server on different hardware. In this case, the new machine needs a configuration that is a composite of the old functionality and the new hardware, but we don't have a configuration like this available to clone.

Most applications of cloning use some process to subsequently customise the clone. In the simplest case, cloning a new machine for a cluster requires that the IP address of the clone is different from the original. But this process can easily get out of hand—it is not uncommon to see large and unmanageable post-cloning scripts.

1. At least, those which are deemed ready for production.
2. That is, starting with an empty disk or one that has no (useful) OS pre-installed.

LCFG can be used as a very effective post-cloning script. The golden copy should be a minimal OS image, including an LCFG client. To create a new machine, this image is cloned. When the new machine first boots, LCFG runs to customise the software and configuration according to the profile. This lets us create very different configurations from the same image. By mixing the header files in the machine profile, we can even create, for example, a new Linux mail server with the same functionality as our old Solaris mail server.

Many tools can be used in conjunction with LCFG in this way, but there are a number of disadvantages: in particular, when LCFG is used on top of an existing clone process, certain parameters are beyond its control. For example, the disk partitioning of the machine will already have taken place by the time LCFG first runs, so the benefits of managing this in LCFG are lost.

Under Linux, LCFG provides an installation mechanism (the LCFG *installroot*) that allows it to take control of the whole installation process. In this case, there is no image to be cloned. The disk is partitioned according to the LCFG profile, and the software is loaded from packages, as specified in the package list. This process is described in Appendix C.

6. Writing Components

Chapter 2 offered a brief introduction to writing your own component. The amount of code required in that case was very small, partly because the LCFG framework provides a lot of supporting functionality. This chapter covers component writing in more detail and supplies a reference for the framework.

Every "subsystem" of a machine that is configured by LCFG needs a component script to read the resources from the profile and generate the appropriate configuration files and daemon options. If there is a daemon process, the component usually controls the lifecycle of the daemon as well (by starting and stopping it). This allows the component to notify (and perhaps restart) the daemon when the configuration changes. It also allows LCFG resources to control which daemons should be running on a particular machine.

Components obey certain conventions about their output and logging, so that status information from the components can be relayed to the LCFG server for display on the status page, and the logs be made available via the logserver.

If you have a new subsystem that you want to configure with LCFG, you should first consider whether it is necessary to write a new component at all. The `file` component can handle most cases only involving the creation of configuration files.

If you do need to manage a daemon or perform more complex processing, you will probably need a custom component. In that case, it is worth looking first for a similar component you can use as a basis—although you should be aware that not all components on the Web site will be exemplars of production code.

Having decided to write your own component, you will need to:

- Be clear about the scope of the subsystem that the component is intended to manage. It is important that there be no overlap between components. For example, individual configuration files or daemons should be managed by one component only.[1] This is a very important principle.
- Choose a language.
- Create a schema file with the types and defaults for the resources to be used.
- Create skeleton code for the component with `lcfg-skeleton`.
- Write code for the `configure` method (see Section 6.2) to create the necessary configuration files from the LCFG resources.
- If a daemon is involved, write code for the methods to manage its lifecycle.
- Code any other methods that may require special treatment.
- Install the component on the client, and the schema file on the LCFG server.

1. Of course, the resources for that component may well come from many different sources.

The LCFG "buildtools" (see Appendix B) is a set of scripts and makefile targets that are useful when building components. They are not strictly essential, and it is likely that they will be replaced at some time in the future, but they currently provide some very convenient functionality. The rest of this chapter assumes their use.

Choosing a Language

The LCFG framework provides support for shell (bash) and Perl components. Using other languages for the component itself is probably not advisable, because this would involve interfacing to (or duplicating) the framework. Of course, components often call utilities written in other languages. To some extent, the choice of language is a personal decision. However, different languages are suited to slightly different applications:

- If you need to actually write a new daemon process (as opposed to simply managing some external daemon command) and the daemon can be written in Perl, then a Perl component is highly recommended. The Perl framework provides a mechanism for communicating configuration changes directly to a running daemon (and for reporting messages directly into the LCFG status system). The component `lcfg-perlex` is a minimal example of a Perl component.
- If the component is very simple and just creates a few configuration files, a shell component is probably most appropriate. The component `lcfg-example` is a minimal example of a shell component.
- If the component is intended to manage a pre-existing daemon, a shell component is usually sufficient. The component needs to start and stop the daemon, notify configuration changes, and ensure that any output from the daemon is routed to the LCFG logging and monitoring system. If access to the C source code of the daemon is available, routines from the framework C library can be added to the daemon itself to handle status reporting.
- The shell framework makes the component resources available directly as shell variables. In some cases (especially when importing large spanning maps), the number of these variables exceeds the shell limit. Perl components are more suitable for handling large numbers of resources.

6.1 Schema Files

Every LCFG component requires a *schema file* to define the schema for its resources. This supplies:

- The list of valid resource names for the component.
- Information on the structure of any list resources.
- Validation predicates (*types*) for resource values.
- Default values.

Simple Resources

Simple resources are declared by specifying their name and default value. For example:

```
ipaddr 129.215.65.78
```

The resource is assumed to be of type *string*, so no validation is performed when the resource is compiled.

Built-in Types

Resources may have a type specified. In this case, the resource values are validated at compile time and, in some cases, transformed into a canonical representation. The types currently supported are:

integer: Validated as an integer.

boolean: Validated as a boolean. Several formats are accepted (e.g., yes and no), and these are all transformed into the canonical true or false. The client translates these values into non-null and null strings so that they can be tested easily from shell scripts.

string: This is equivalent to having no type specification.

Type specifications have the form:

```
@name %type
```

For example:

```
@debug %boolean
debug yes
@interval %integer
interval 10
```

String Validation

String resources may have arbitrary validation code attached. For example:

```
@url %string(http url): /^http:/
url http://www.lcfg.org
```

If the value does not satisfy the validation predicate, the name in brackets is printed as part of an error message. The file validate.h provides a number of standard validation predicates (see Figure 6.1). Arbitrary Perl code can be used for validation, but care is required, since this executes in the context of the compiler (albeit in a “safe” module).

vENUM(L)	The value is a member of the token list L.
vINFILE(F)	The value matches a line in the file F.
vIPADDR	The value is a valid IP address.
vIPADDRLIST	The value is a (space-separated) list of valid IP addresses.
vHOSTNAME	The value is hostname present in the DNS at the time of compilation. Note that this will not automatically be revalidated if the DNS is subsequently changed.
vHOSTLIST	A (space-separated) list of valid hostnames.
vURL	A URL.

Figure 6.1: Standard Validation Macros

Tag Lists

LCFG resources support a single compound datatype called a *tag list*. As noted earlier, this has some similarity to both the lists and the hashes usually found in programming languages; the elements are ordered, but they can also be identified by a named tag.

The notation for declaring tag lists is unfortunately rather awkward[2]—for example, we saw these resources earlier:

```
kdm.menu file quit saveas
kdm.mitem_file File
kdm.mitem_quit Quit
kdm.mitem_saveas Save As
```

The corresponding type declaration would be:

```
@menu mitem_$
mitem_$
```

Notice that this allows us to provide default values for both the list of tags and the individual elements:

```
@menu mitem_$
menu First Second
mitem_$ A Menu Item
```

A tag list may have more than one resource per element:

```
@devices     dev_$ perms_$
dev_$        /dev/null
perms_$      0644
```

An instance of this list might look like:

```
foo.devices          knife fork
foo.dev_knife        /dev/knife2
foo.perms_knife      0655
foo.dev_fork         /dev/fork
foo.perms_fork       0600
```

The notation for declaring multi-level tag lists can be particularly confusing, and several different styles are in use. For example, the second-level resource keys may contain only a single tag:

```
@disks dopartition_$   partitions_$
disks
dopartition_$
@partitions_           pdetails_$
partitions_$
pdetails_$
```

Or they may contain the tags from both levels:

2. This is a consequence of evolution from the simple list markup convention used in the original LCFG implementation.

```
@modules     entries_$
modules
@entries_$   entry_$_$
entries_$
entry_$_$
```

Some old components do not provide an explicit tag list; they assume an implicit tag list of 1..N, where N+1 is the lowest integer for which no matching resource exists. This is not recommended, but it can be simulated for compatibility by specifying a # in the tag list. For example:

```
@rules       rule_$
rules        foo #
rule_foo     R1
rule_1       R2
rule_2       R3
rule_4       R4
```

This would generate resources corresponding to an implicit tag list of:

```
foo 1 2
```

Notice that rule_3 is ignored. There is a limit of 100 on these enumerated tags.

List Sorting

Typically, the value of a list resource is not fully specified in any single file but is built up from declarations spread across several header files, representing different aspects. For example, the list of components that is started at boot time is usually defined by the resource boot.services. The basic site header file normally defines a default list of services, but optional header files will add other services, such as a Web service or a database service (e.g., using mADD).

In some cases, as in the boot order above, the (partial) order of the items in the list is important. If the optional header files simply append items to the end of the list, their order depends on the ordering of the header files, and this can vary.

The LCFG compiler provides a mechanism to sort the items of a list automatically, according to precedence constraints. For example:

```
boot.services a b c d e f
boot.order_a >c >d <e
boot.order_c >d
```

The boot.services list will be (topologically) sorted so that a comes (not necessarily immediately) after c and after d, but before e. c will also come after d. The order of unconstrained items in the sorted list is not defined. (Clearly, it is possible to specify contradictory constraints, but that will generate a compile-time error.)

The name of the resources containing the ordering constraints must be specified in the definition of the list resource. In addition to specifying resources of the current component, it is also possible to specify that the ordering resource comes from some other component; this is very useful in cases such as the boot.services example, because

additional components can be added and their ordering constraints can be included in their own schema file without any changes to the header files or the boot defaults. For example:

```
@boot.services foo_$ order_$ ; order_$ $.bootorder
```

In this case, the ordering constraints for the component b can be specified either in boot.order_b or in b.bootorder, or in both.

Occasionally, it may be useful for the component to know the explicit ordering constraints for the items, as well as the sorted list. This would be necessary, for example, for the boot component to determine whether certain services could be started in parallel. The compiler can store the final constraints in specified resources. For example:

```
@boot.services order_$ after_$ before_$ ; order_$ >after_$ <before_$
```

This definition will cause the compiler to generate resources such as after_a, which contains the list of items that must come after a, and before_a, which contains the list of items that must come before a. If this definition was used with the resource values above, the following values would be generated:

```
after_a = e
before_a = c d
after_c = a
before_c = d
after_d = a c
before_e = a
```

Spanning Maps

Spanning maps are a very important feature of LCFG. They allow resource values to be collated from many different machines and integrated into the profile of some other machine. This allows you to create components that automatically manage the relationships between machines.

For the user of a spanning map, the process is almost transparent. But the component author needs to create the appropriate schema files. In all, four configuration files are involved—the subscriber and publisher source files (created by the user) and the subscriber and publisher schema files (created by the component author). This is best illustrated by an example:

- The schema file for the dhcp client component specifies which resources are to be exported:

  ```
  name
  mac
  ...
  @map %publish: name mac
  map
  ```

  This says that the resources name and mac are to be published to the spanning map whose name is given by the map resource.

- The dhcp *client* source files specify only the map name to which the resources should be published (and, of course, the values of the resources themselves):

  ```
  name foo
  mac 1.2.3.4.5.6
  ...
  map dhcp/cluster27
  ```

  The user need not be aware of which resources are being published.

- The dhcp *server* schema file specifies the name of a list resource into which the map entries will be imported. The fields of the list resource should correspond to the resource names that will be published to the map:

  ```
  @clients name_$ mac_$
  clients
  name_$
  mac_$
  ...
  @map %subscribe: clients
  map
  ```

  This specifies that a list of all the clients publishing to the map named in the map resource should be generated and stored in the clients resource. For each client, the values of the name_*client* and mac_*client* are generated from the values of the corresponding client resources.

- The dhcp *server* source file need only specify the map to which to subscribe:

  ```
  map  dhcp/cluster27
  ```

The result of this is that the clients resource in the server profile will include the data from all the clients that have published to the specified map. The list tags are the node names of the clients. This is equivalent to having manually created the following:

```
clients client1         client2 ...
name_client1            foo
mac_client1             1.2.3.4.5.6
name_client2            bar
mac_client2             6.5.4.3.2.1
...
```

If any of the published resources in a node is changed, all nodes that subscribe to the map are recompiled automatically. A node may publish and subscribe to the same map.

Resources of type %publish and %subscribe may list multiple maps, allowing resources to be exported and imported from several different maps. The same resources can be exported by several different %publish resources, and it is possible to export a resource with a different name:

```
@map %publish: name ether=mac
```

This will export the value of the resource mac with the name ether.

Resources from different components can be published to the same map as long as the field names of the subscribe resource include the names of all the published resources. (References can also be used to collate values from multiple components.)

If a list resource is published, only the one resource containing the tag names is exported; the sub-resources of the list are not automatically exported.[3] Cross-domain spanning maps[4] require unique (short) node names for the publishers, because the short names are used as the list tags in the imported map.

Common Resources

In addition to the application-specific component resources, most components will want to include the following:

```
#include "ngeneric-1.def"
#include "om-1.def"
```

- The ngeneric resources are described in the lcfg-ngeneric man page. These resources are interpreted by various parts of the LCFG system itself and control logfile rotation, configuration dependencies, monitoring and status behaviour, and some other options.
- The om resources are interpreted by om, mainly for authorisation (see the manual page for LCFG::Authorize).
- Components should also include a schema resource specifying the version of the schema that they require. This allows for schema changes which are not backwards-compatible.

Extending Existing Schema

Since the schema file is passed through the C preprocessor, it is possible to extend existing component schema by including the schema files of those components. Overrides and mutations are supported so that the inherited resources can be changed if required. For example, the ngeneric resources for log rotation can be extended:

```
!ng_logrotate mEXTRA(tr)
ng_logrotate_tr copytruncate
```

It is even possible to mutate the type defined by an included component to add additional fields to a list record or to add additional validation. The following example creates a local version of the client schema which adds additional validation to the LCFG server URL:

```
#include "client-2.def"
!schema mSET(local-2)
!@url mSET(%string(interval): /^http:foo.com/)
```

Notice that the header files containing the macros for mutation and validation (e.g., the mEXTRA) should be included explicitly if they are required:

3. This is an implementation restriction we would like to remove.

4. That is, a spanning map that publishes resources collated from machines in different domains. In this case, the short hostnames used as resource keys may not be unique.

```
#include "mutate.h"
#include "validate.h"
```

Pseudo-Nodes

Sometimes it is useful to create source files that do not represent real machines. These can used as either publishers or subscribers to spanning maps. For example:

- An inventory source file can be used to collate all the inventory information published in the machine source files. By default, the inventory is available as an XML profile, but a plug-in LCFG server module can be used to generate this in a different format.
- Source files can be created to represent printers, and the information needed by the print servers can be published to a spanning map. The print servers can then subscribe to the spanning map to get the list of printer names and attributes.
- As a combination of both, a pseudo-node can subscribe to the printer information and feed the resources into LDAP using a plug-in module.

6.2 The Component Framework

The LCFG framework consists of three main packages:

- lcfg-skeleton is a script to interactively create a complete set of skeleton files for a new component.
- lcfg-utils is a set of utility functions with Perl and shell bindings. This includes functions for manipulating resources, reporting status, etc.
- lcfg-ngeneric provides the "superclass" for LCFG components in Perl or shell. This is a complete implementation of a component, but without explicit code for the core of the methods. This is what allows new components to be written simply by supplying the component-specific code for the methods.

LCFG-Utils

lcfg-utils provides C libraries, Perl bindings, and shell commands[5] for a number of standard functions:

- lcfgmsg is a command-line utility, and LCFG::Utils is a Perl module, both based on the C library liblcfgutils. These routines format and route error and log messages, as well as notifying the client component (and ultimately the LCFG server) of any status changes.
- qxprof is a command-line utility based on the Perl module LCFG::Resources. It copies resources between various formats: resources can be read from the profile, from a file, from the command line, or from the environment. The values can be written to a file or the environment. This is the primary interface to

5. Components can certainly be written natively in other languages, but that requires writing and maintaining a copy of the framework in that language. It is more common to see components that use a shell wrapper whose methods call separate programs in some other language.

the profile. These functions are called automatically by the generic components (see below); it is not usually necessary to call them explicitly.

- sxprof is a command-line utility based on the Perl module LCFG::Template. It takes a flat-text template file and substitutes variable values from LCFG resources. As with qxprof, resource values can be taken from several sources. For most components, sxprof is sufficient to generate complete configuration files directly from LCFG resources without any additional coding. sxprof is the heart of the file component.

lcfg-ngeneric provides *generic* components (for shell and Perl) which act as superclasses for creating component instances. These give the default semantics for the standard methods, including resource loading, locking, error checking, and standard option processing. They also provide additional utility functions and convenient access to the functions in the lcfg-utils library. The shell generic component consists of a file of shell functions which can be sourced by a component shell script. The Perl generic component is a Perl object class which can be subclassed to create a component instance.

Shell Bindings

The ngeneric script provides support for components written in the bash shell. Components simply source ngeneric, which provides a number of useful shell functions as well as default code for all standard methods. The manual page for lcfg-ngeneric describes the available functions.

The Dispatch() function should be called with the command-line arguments. This parses the common options and calls the corresponding method. The absolute minimal component script is:

```
#!/bin/bash
. /usr/lib/lcfg/components/ngeneric
Dispatch "$@"
```

This will support all the standard methods and options, perform locking and logging, and load the component resources. To add application-specific functionality, it is simply necessary to override some of the default methods.

For a method *foo*, Dispatch() calls the Shell function Method_*Foo*(). This performs some generic operations before calling the function *Foo*(), which is normally defined to be empty. Component scripts simply need to redefine the function *Foo*() for any methods they want to support.[6] The generic operations include locking, loading of resources, and some error checking. When the user function is called, the LCFG resources are usually available as environment variables, and the standard options have already been parsed. For example, the component could redefine the Start() function as follows:

```
Start() {
    Info "Starting my component"
    Info "My arguments are $*"
```

6. The function Method_*Foo()* can also be redefined in special cases, although this is discouraged, because it is likely to change the standard method semantics.

```
        Info "My server resource is $LCFG_foo_server"
        Info "The verbose flag is $_VERBOSE"
}
```

- The Info() function is a standard function for displaying informational messages. Functions such as this should always be used, rather than simply "echoing" messages.
- The arguments are those supplied on the command line, following the method name, when calling the component (after removal of any generic options). These component-specific arguments can be used for any purpose.
- The names of the environment variables used to hold the resources are determined by qxprof (see the manual page).
- The exact operations performed before calling the user function depend on the method. These are described in detail on the lcfg-ngeneric manual page.
- The standard options are available as environment variables (see Appendix B).

ngeneric also includes a number of other utility functions which are described later. The manual page (man lcfg-ngeneric) provides further details. The source code is also quite simple to read, and the lcfg-example component provides a complete simple example.

Perl Bindings

The Perl module LCFG::Component provides a superclass that can be inherited to create pure-Perl components. This module provides all the functionality of the ngeneric shell functions, including methods, utility functions, and variables. The corresponding minimal Perl component is:

```
package LCFG:: Foo;
@ISA = qw(LCFG::Component);
use LCFG::Component;
new LCFG:: Foo() -> Dispatch();
```

The component methods are Perl member functions, and the resources are passed as Perl data hashes. A simple user-defined Start() function might look like:

```
sub Start($$@) {
        my $self = shift;
        my $res = shift;
        my @args = @_;
        $self->Info("Starting my component");
        $self->Info("My arguments are ."join(' ',@args));
        $self->Info("My server resource is ."
                        $res->{'server'}->{VALUE});
  $self->Info("The verbose flag is ."$self->{_VERBOSE});
}
```

- The methods are Perl object methods.
- The resource hash contains resource meta-information as well as values. See man LCFG::Template for details of the format.

- The standard options are available as member variables.

The LCFG::Component module includes utility functions similar to those of ngeneric, as well as the I/O handling functions, and some additional routines for supporting daemon components. The manual page (man LCFG::Component) provides further details on available variables and functions. The lcfg-perlex component provides a complete simple example.

The Template Processor

The template processor is a powerful utility for creating configuration files by substituting LCFG resource values into template variables. It supports conditionals and iteration based on LCFG resource lists. This can be used to create most configuration files very easily, with no additional code.

The command-line utility sxprof is based on the Perl module LCFG::Template, so identical template files can be processed either from Perl or from the shell. Typically, sxprof is called to read a template and substitute the values of LCFG resources, creating a new configuration file. The values of the resources are usually obtained from the environment (where they are placed automatically by the generic component):

```
sxprof -i component template outfile
```

The format of the templates is best illustrated with some examples. The most basic usage is the substitution of a simple resource value—for example, to create a sendmail.cf file and substitute the value of the mail relay from the LCFG relay resource:

```
...
DH<%relay%>
...
```

Iteration over tag lists is supported automatically, so multiple lines can be generated for list resources:

```
fstab.partitions hda1 hda2
fstab.mnt_hda1 /
fstab.args_hda1 ext2 defaults 1 0
fstab.mnt_hda2 swap
fstab.args_hda2 swap defaults
```

Using the template:

```
<%for: item=<%partitions%>%><%\%>
/dev/<%item%> <%mnt_<%item%>%> <%args_<%item%>%>
<%end:%><%%>
```

yields:

```
/dev/hda1 / ext2 defaults 1 0
/dev/hda2 swap swap defaults
```

The syntax may seem awkward, but this is largely due to the rather obscure delimiters.[7] The evaluation process is really quite straightforward. For example, during the first iteration of the above loop, the variable item is assigned to the value of the first tag from the list resource partitions (i.e., hda1). The second field of the fstab is set to <%mnt_<%item%>%>, which evaluates to <%mnt_hda1%> and hence swap.

The exact character sequence (including newlines) appearing outside the <% and %> characters is copied to the output. Hence the use of the <%\%> symbols, which are used to prevent unwanted newlines appearing in the output.

The template processor also supports:

- File inclusion (<%include:%>).
- Conditionals on the value (<%if:%>) or the existence (<%ifdef:%>) of a resource.
- Evaluation of arbitrary shell (<%shell:%>) or Perl (<%perl:%>) expressions and the substitution of their output.
- Arbitrary variables which can be set from the command line or can be the results of evaluating some other expression.
- Insertion of resource derivations as well as values (<%#variable%>). This is useful for comments in the generated file.
- Comments in the template which are not copied to the generated file (<%/_%>...<%_/%>).

See the LCFG::Template man page for the full details.

When evaluating conditionals, the empty string is considered false and all other values (even 0) are considered true. This is consistent with the LCFG client's treatment of resources that are declared as boolean: the client maps any representation of false onto a null string so that it may be tested more easily with the shell test function.

The return status from sxprof also indicates whether the resulting output file has been changed by the substitution. This is very useful in components that manage daemons, since the daemon may need to be notified or even restarted when the configuration changes:

```
sxprof -i foo template output
status=$?;
[ $status = 2 ] && LogMessage "configuration changed"
[ $status = 1 ] && Fail "failed to substitute template"
```

A similar process can be used to automatically create command-line arguments for a daemon and to force a restart if they have changed:

```
sxprof -i  foo - argfile <<EOF
<%if: <%debug%>%> -D '<%debug%>'<%end:%><%%>
<%if: <%verbose%>%> -v<%end:%><%%>
<%if: <%xmldir%>%> -x '<%xmldir%>'<%end:%>
```

7. The delimiters can be changed with command-line arguments, but the default is deliberately rather obscure, to reduce the chance of misinterpreting any characters that are a literal part of the template file.

```
EOF
if daemon is running ...
    if [ $? = 2 ]; then
            stop daemon
            daemon 'cat  argfile'
    fi
fi
```

If changes to certain parts of the template are insignificant (e.g., comments), the text can be included inside the delimiters <%{%> and <%}%>. This will prevent changes to this text from causing a return status of 2, which would lead to an unnecessary notification of the daemon.

Utility Functions

lcfg-utils provides the following utility functions:

- Do(): The arguments to this function are executed as a shell command. If the debugging option (-D) is set, the command is also printed as a debug message. If the dummy option (-d) is set, the command is printed without being executed.
- IsStarted(): Returns true if the component is currently started.
- RequestReboot(): Sets a flag in the status display indicating that the node requires a manual reboot.
- ClearReboot(): Clears the reboot flag.
- SetPwrCycle(): Sets a flag in the status display indicating that a power shutdown has been scheduled.
- ClearPwrCycle(): Clears the power shutdown flag.
- SaveStatus(): Saves resources from the environment to the status file.
- LoadStatus(): Loads resources from the status file into the environment.
- LoadProfile(): Loads resources from the profile into the environment.
- Lock(): Locks the component (blocking).
- Unlock(): Unlocks the component.

SaveStatus() is automatically called by the generic component after successful completion of a configure method to save the configured resources. These resources are automatically loaded again (using LoadStatus()) at the start of methods such as run so that the resources in the environment represent the values that are currently configured—these will be different from those in the profile if a previous configure operation failed.

Lock(), Unlock(), and LoadProfile() are also called by the generic component and do not normally need calling explicitly.

Component Output

Component scripts often run at boot time and at other times when error messages may go unnoticed or verbose output might obscure other important messages. Components

should restrict output to a few well-defined messages, written to stderr. More verbose information should be written to the logfile. Messages should only be generated on stdout when that is the purpose of the method (e.g., status).

At boot time, messages should be formatted to conform to the standard system boot message format.

Ngeneric provides the following functions:

> OK(): This is called automatically by the framework on successful completion of a method.
>
> Fail(): The component should call this function with an error message to abort the method. The failure is notified to the LCFG server for indication on the status display, and it is logged in the logfile.
>
> Error(): The component should call this function with an error message. The error is notified to the LCFG server for indication on the status display, and it is logged in the logfile.
>
> Warn(): The component should call this function to print a warning message. The warning is notified to the LCFG server for indication on the status display, and it is logged in the logfile.
>
> Info(): The component should call this function to print an informational message, usually only when requested with a verbose option. The message is also logged in the logfile.
>
> LogMessage(): The component should call this function to print a message to the logfile.
>
> Debug(): The component should call this function to print a debug message, usually only when requested with a debug option.
>
> StartProgress(): The component should call this function to print a message followed by a progress indicator. The function Progress() is called at intervals to advance the indicator, and the function EndProgress() should be called when the operation is complete.

Note that calling Fail() (e.g., during a Start()) will abort the method (so the component will not be considered as started). Error() can be called to indicate a problem without aborting the method call.

The following example shows the recommended way of handling long error messages and of debugging messages so that they do not clutter the display. The environment variables for the standard options are described in Appendix B. The verbose option can also be enabled on some platforms by holding down the shift key when the component method is called.

```
[ -n "$_DEBUG" ] && Debug "Debug message"
if [ -n "$_VERBOSE" ] ; then
    Error "A long error message"
else
```

```
        Error "Short message (see logfile)"
        LogMessage "A long error message"
    fi
```

The above functions support the coloured text used by Red Hat during startup. Newlines embedded in arguments are handled correctly. The C library lcfgutils provides access to these functions from C, allowing them to be called directly from C helper programs.

The generic component redirects the standard output and error descriptors to the logfile, so all messages not produced by the above functions will appear there. If a component needs to print to the standard output or error (e.g., as part of a status method), the descriptors 11 and 12 can be used:

```
cat mylogfile >&11
```

Command return status should be checked and Fail() called to abort the component when necessary.

Handling Logfiles

The generic component defines the variable $_LOGFILE to be the name of the standard component logfile. Standard output and error descriptors are redirected to the logfile so that the component may simply write to stdout to append messages to the logfile. The function LogMessage() generates time-stamped and formatted messages, which are usually preferable.

Sometimes a component may need several logfiles for different purposes. They should be named by adding extensions to the standard logfile name; this makes the logfile visible (when permitted) by the logserver component and allows the logfiles to be easily rotated using the standard log rotation files.

Logfiles with the standard extensions .err and .warn are created automatically by the LCFG event routines. These files contain any error and warning messages generated by the component, and their presence is detected by the status reporting system and used to display error and warning icons on the status display. These files are deleted only by the Reset() method (or a reboot) so that error messages will not be removed before they are manually acknowledged.

The generic Configure() method creates a logrotate file to cycle the logfiles at various intervals. The logrotate file is created by passing a default template through the template processor. This allows resources to be used to customise the log rotation:

ng_extralogs: A list of extensions for any additional logfiles to be rotated.

ng_logrotate: A list of tags representing additional lines to be inserted in the logrotate file.

ng_logrotate_tag: The logrotate line corresponding to the tag.

If even more control over the log rotation is required, the component can include a custom template in /usr/lib/lcfg/conf/component/logrotate.

The standard log rotation file calls the logrotate method on the component after the logfiles have been rotated. This can be used where necessary to force daemons to close and reopen their logfiles.

Option Processing

The generic component parses some standard options and makes them available in the following variables:

$_DUMMY (-d): The component actions are printed but not executed.

$_DEBUG (-D): Prints debugging information.

$_NOSTRICT (-n): Certain warning and error messages are suppressed. For example, trying to stop a component that is not started will normally generate a warning message. If this option is used, the warning is not generated.

$_QUIET (-q): No messages are printed.

$_TIMEOUT (-t): If a component is already executing, calls to most methods will usually block until the existing instance terminates and releases the lock. This option specifies a timeout so that the current call will terminate after timeout seconds if the lock cannot be obtained. Certain method calls do not lock (see the list above). Locks can be broken using the unlock method.

$_VERBOSE (-v): Additional messages are printed. Holding down the shift key when a component method starts executing will also enable this option on some platforms. This is useful at boot time to enable more verbose logging on certain components.

Component methods must parse any method-specific options explicitly. For example:

```
Run() {
    while getopts ":x:y" arg ; do
            case $arg in
                    x') Info "option x is $OPTARG" ;;
                    y') Info "option y specified" ;;
                    '?') Fail "bad option ($OPTARG)" ;;
            esac
    done
    ...
}
```

Standard Variables

The generic components provide some other standard variables:

$_COMP: The component name.

$_LOCKDIR: The lock directory name.

$_LOGFILE: The logfile name.

$_OKMSG: The generic components print the message given by this variable on successful completion of a method. This can be modified to add small amounts

of extra information (but should not be used for long messages!). For example: _OKMSG="$_OKMSG (custom message here)"

$_ROTATEDIR: The directory for logrotate files.

$_RUNFILE: The run file. This file is created as a marker to indicate that the component has started.

$_STATUSFILE: The status file name. This contains the values of the resources set at the last successful reconfiguration.

Locking

The generic component assumes that most methods are not re-entrant; a per-component lock is used to block method calls if some other method is currently executing. The descriptions in section 4.1 note those methods that are not locked by default.

The functions Lock() and Unlock() call the program lcfglock to take and release the locks. User-supplied method code can call these functions to lock custom methods or methods that do not normally lock by default. By (conditionally) calling Unlock() before Dispatch() it is possible to disable the default locking of the standard methods, although this is not recommended—the caller should use the -t option or call the unlock method to break existing locks.

The variable $_TIMEOUT is set from the generic -t option. This can also be set explicitly by component code to define a default lock timeout. The variable $_LOCKDIR is set to the name of the directory used to hold the lockfiles.

The Configure Method

The configure method is the most important method. It is called whenever the component resources are changed. The component script should update the configuration files to reflect the new resource values. If any daemons are currently running, the component should do whatever is necessary for the daemons to recognise the updated configuration.

The example component shows a typical configure method:

```
Configure() {
    # Use sxprof to create the config file:
    /usr/bin/sxprof -i $_COMP template config-file
    status=$?
    # Check status
    [ $status = 1 ] && Fail "sxprof failed (see logfile)"
    # Return if no change
    [ $status = 2 ] || return
    # Check if the daemon is running.
    # If so notify it of any changes (if necessary)
    LogMessage "configuration changed"
    ...
}
```

A resource may change for several reasons, including a change to the specification on the LCFG server or a local change of context. The machine may not even be connected to the network at the time the change occurs, and the component should not need to know the reason for a particular change.

Sometimes it is useful for the component to know the previous values of any resources that have changed. The qxprof command can be used with the -r option to read these from the status file. Providing a different prefix for the loaded variables enables both sets (the previous and current values) to be present in the environment at the same time:

```
_NG_GLOBSTAT=‘set +o |grep noglob‘ ; set -f
eval ‘/usr/bin/qxprof -p old_LCFG_%s_,old_LCFGTYPE_%s_ \
            r $_STATUSFILE -e $_COMP 2>>$_LOGFILE ; \
    echo export _STATUS=$?‘ 2>/dev/null
eval $_NG_GLOBSTAT
[ “$_STATUS” != 0 ] && Warn “failed to loadresources”
```

Immediate update of configuration changes is not always appropriate. The component must decide whether certain changes should be deferred; for example, if a user is currently logged onto the console, the component that manages the display will defer (until the user logs out) updates that require the display server to be restarted. Some changes can still be difficult to schedule; for example, changes to disk partition sizes will not normally be implemented until a rebuild operation is initiated manually.

Two standard resources are interpreted by the client component to determine when to call a component’s configure method:

ng_cfdepend: This resource is interpreted by the LCFG server (and, ultimately, by the client). It is used to determine which components should be reconfigured when resources change. The resource should include a list of dependencies of the form >*component* or <*component*. In the first case, the specified *component* will be reconfigured whenever the resources of this component change. In the second case, this component will be reconfigured whenever the resources of the specified *component* change. Normally, this resource will be set to <*self* so that the component’s configure method is called whenever its own resources change.

ng_cforder: This resource is interpreted by the LCFG server. It is used to generate the client.components resource, which specifies the order in which components should be reconfigured after a configuration change. ng_cforder specifies a list of constraints on the order in which the components are reconfigured. A constraint of the form >*component* means that this component must be configured after *component*. Similarly, <*component* means that this component must be configured before *component*. A runtime error will occur if the constraints specify a loop.

Managing External Daemons

In addition to creating configuration files, many components also manage one or more daemons. This is not essential—daemons can simply be started and stopped using the normal init process. But using an LCFG component makes it easier to notify the daemon when the configuration changes and to set command-line options from LCFG resources. It is often possible to create an init script (or use an existing one) and just call this from the LCFG component methods:

```
Start { /etc/rc.d/init.d/foo start }
Stop { /etc/rc.d/init.d/foo stop }
```

Typically, the Configure() method would simply call Stop() followed by Start() to restart the daemon whenever the configuration changed (see the example in section 2).

If there is no existing init file or a more complex startup process is needed, it may be more convenient to simply stop and start the daemon directly from the LCFG component. The shell generic component provides a Daemon function to perform some I/O redirection and other preliminaries before forking a background process. The component will probably want to store the process ID so that it can be located later to stop or notify the daemon:

```
Start {
    Daemon "foo 'cat argfile' 2>/dev/null"
    client_pid=$!
    [ -z "$_DUMMY" -a -z "$client_pid" ] && \
            Fail "failed to start foo (see logfile)"
    echo $client_pid >$PIDFILE
}

Stop {
    client_pid='cat $PIDFILE 2>/dev/null'
            [ -n "$client_pid" ] && [ -e /proc/$client_pid ] && \
                    Do "kill -INT $client_pid"
            rm -f $PIDFILE
}
```

Since the Configure() method is called as part of the generic Start() method, command-line arguments can be constructed (from the resources) in the Configure() method (see below). This allows the Start() method to simply retrieve them from the argfile, as shown above.

After starting or stopping a daemon, before exiting the method it is useful to check that the operation has been successful. This might, for example, involve a delay loop that polls for the existence of a process after sending it a termination interrupt. The standard sendmail init files, for example, sometimes exit before the sendmail process has actually terminated. Immediately calling a subsequent init start (as one might do in a Configure() or Restart() method) will fail intermittently because there is already a process running. The Stop() method of the mailng component is a good example of how to handle this situation correctly.

It is important to make a distinction between a component being started and the corresponding daemon being started. The component is considered started after a successful call to the Start() method and before a successful call to Stop(). This is the status reported by the IsStarted() function. Starting daemons correctly and detecting errors can be hard because the daemon may fail asynchronously after it has apparently started successfully. It is sometimes useful to sleep for a short time after starting a daemon before checking that it is still running; this helps to detect any obvious failures that might occur during daemon startup. Subsequent failures can only be detected by regular polling, perhaps using the Monitor() or the Run() method, called from cron to check the health of the daemon and report or correct any failures.

The standard output (and error) channels from the component (and hence the daemon) are redirected to the logfile, so all daemon messages will appear there. However, error messages from the daemon will simply appear in the logfile without generating LCFG error events (i.e., the errors will not appear on the LCFG status display). If the daemon source code can be modified, then explicit LCFG event routines can be added using the lcfgutils C library.

Writing Daemons in Perl

The Perl generic component can be used to create components without daemons or components that manage external daemons, as described above. In addition, it provides support for writing components that are themselves daemons; that is, the component process forks in the Start() method in order to leave a copy running in the background (both of these alternatives are illustrated in the example component lcfg-perlex). This has the advantage of providing a much tighter coupling between the running daemon and the LCFG framework: configuration changes are notified directly to the running daemon, which can usually handle most changes on the fly without requiring a restart. The process is as follows:

- The Start() method should perform any initialization and then call StartDaemon(), which forks. The parent copy returns and hence exits the Start() method. The child calls the user-supplied DaemonStart() function, which forms the main loop of the daemon.
- The Stop() method should call StopDaemon(). This signals the running daemon process and automatically calls the user-supplied DaemonStop() function, which is responsible for terminating the main loop of the daemon and exiting.
- The Configure() method should call ConfigureDaemon(). This signals the running daemon process and automatically calls the user-supplied DaemonConfigure() method. The new values of the resources are read into the daemon process automatically and provided to DaemonConfigure() as arguments. In many cases, the daemon process can simply store the resources in global variables or perform some simple reconfiguration that allows it to adopt the new values without restarting.
- All the standard utility functions are available to the daemon process, so error reporting and other logging can use the standard functions and events are reported immediately to the LCFG server.

6.3 Testing

LCFG components are simple scripts. In theory, it is possible to test them just by executing the script with the appropriate method as an argument:

```
$ ./mycomponent start
```

In practice, there are a number of problems with this:

- The resources are obtained from the profile of the host running the test. These resources might not exist in the profile, or you might want to use different values during testing.
- The ngeneric component requires root permission to write to several logfiles and status files. You probably don't want to write to these live files during testing.
- It is likely that the component-specific code will also need to write to root-owned configuration files or make other changes to the live system.
- It is possible that the component will need to start daemons or perform other root actions that you might not want to do on the live system during testing.

The LCFG buildtools (see Appendix B) provide support for all the cases discussed below.

Test-time Status Files

If the current directory contains a file called test.mk, the buildtools will automatically define the variables @TESTSHELLV@ and @TESTPERLV@. These variables contain redefinitions for all the system status and logfiles used by the generic component. For shell components, the variable should be included when sourcing the ngeneric component:

```
@TESTSHELLV@ . @LCFGCOMP@/ngeneric
```

For Perl components, the variable should be used when creating the component object:

```
new LCFG::PerlEx(@TESTPERLV@) -> Dispatch();
```

By default, the private files are created under a subdirectory called TEST in the current directory. The pathnames for all the individual files can be changed by assigning different values to the corresponding buildtools variables. The defaults are defined in lcfg.mk.

The test.mk file is normally included, along with the other source files, in CVS for the module. The buildtools will not package this file for distribution, so packaged and distributed components will use the live pathnames. However, any attempt to run the component in the working directory will use the test pathnames: this is why you were asked to delete this file earlier, so that the component would be built with live values.

Test-time Resource Values

If the buildtools variable @TESTRES@ is defined, it is assumed to be the full pathname of a file containing resource values. When the file test.mk exists, these values will be used instead of any values obtained from the profile. The format of the resource file should be suitable for reading with qxprof -r (this is the same format as generated by qxprof -w).

Conventionally, the @TESTRES@ variable is defined in the test.mk file.

Test-time Configuration Files

Any buildtools variable definitions in test.mk will take precedence over definitions in config.mk or in any of the standard buildtools symbol files. By defining names for live configuration files in config.mk and corresponding test-time names in test.mk, components can be tested in the working directory without writing to the live files.

Test-time Command Execution

The buildtools define the variable @TESTING@ when the test.mk file is present. This can be used in the component code to take different actions during testing. For example, a debug message may be printed, rather than starting a live daemon that requires root privileges.

The Do() function is also useful for testing. Privileged system operations should be called using this function. For example:

```
Do "/etc/init.d/rc.d/sendmail start"
```

In normal operation, this will execute the specified command. However, if the component is called with the -d option, a debug message will simply be printed instead.

Test Installation

At some point it will be necessary to test the component in the live environment. The buildtools target devrpm builds an RPM from the files in the working directory. This RPM can be installed and tested on the current system before checking in the code and building a production RPM. Test RPMs should never be shipped to production systems, since the code is not guaranteed to exist under CVS.

If the nsu command is available, the buildtools target devinst can be used to create the development RPM and install it on the current system with one command.

Summary

In summary, the following steps should make component testing straightforward:

- Write the component to include @TESTSHELLV@ or @TESTPERLV@, as described above.
- Create a file containing resource values to be used during testing. Define @TESTRES@ to be the name of this file.
- Define the names for system configuration files in config.mk and provide test-time names for them in test.mk.
- Use Do() to execute any commands that require system privileges; test the component by using the -d option.
- Use the buildtools devinst target to install a test copy on the current live system.

6.4 Packaging and Installation

To use an LCFG component, the code needs to be installed on all the appropriate clients and the schema file needs to be installed on the LCFG server. LCFG does not mandate any special way of doing this, but mature LCFG sites will probably use LCFG itself to manage the distribution and installation of the packages.

The details of the packaging and installation process will depend on the method used. This section brings up a few general issues you may want to consider.

Reconfiguring on Component Upgrade

When a component is upgraded, there will often be some kind of a change that requires a reconfiguration. You can force this by using an RPM post-install script in the specfile:

```
%post
if [  -x @LCFGCOMP@/@COMP@ -a \
              -f @LCFGTMP@/@COMP@.run ] ; then
    echo reconfiguring @COMP@ component
    /usr/sbin/daemon @LCFGBIN@/om @COMP@ configure -- -f
fi
exit 0
```

In most cases the configure method will not restart a daemon, for example, unless the resources have changed. However, in this case, we do want to force the daemon to restart, since the daemon code may have been upgraded. The -f flag is not interpreted by the framework in any way, but it is a convention which should be handled by the configure method to force a complete reconfiguration even if the resources have not changed. If the configure method does not expect any other special flags, the following code would be typical:

```
while [ -n "$1" ] ; do
  [ "$1" = "-f" ] && _RESTART=1
  shift
done
sxprof ...
[ $? = 2 ] && _RESTART=1
[ $_RESTART = 1 ] &&  Restart the daemon
```

Note that the configure method will run in the context of an RPM install. This requires some care over the environment when restarting daemons.

Installing

The following steps are required to install and use a newly created component:

- The component code must be installed on the client.
- The schema file must be installed on the LCFG server (buildtools creates a separate RPM for the schema file, and this should be installed using the appropriate process).
- Any clients using the component need to specify the appropriate schema

version (usually this is included in some header file):

profile.version_*component version*

- The component should be added to the component list of the appropriate clients (usually this is included in some header file) :

  !profile.components mADD(*component*)

- If the component is to be started at boot time and the boot component is being used, the new component should be added to the boot list:

  !boot.services mADD(lcfg_*component*)

- Some other boot resources may need setting to control the order and run levels.

7. Finally . . .

You should now have a good understanding of what is involved in implementing LCFG at your site, including both the benefits and the difficulties. If you decide to make use of LCFG, you will probably find that the organisational issues described in Chapter 5 become more apparent: these are the real, practical challenges of system configuration.

The LCFG Web site and wiki hold the latest code and more detailed documentation. Corrections and updates for this booklet will also appear on the Web site. We would be glad to have your experiences and contributions via the mailing list or the wiki.

7.1 LCFG Developments

The original LCFG concept and the current implementation have been very successful in real production environments. The components themselves are small and self-contained, and they have evolved well. But the LCFG server code is now rather old and difficult to extend. Some of the newer features, such as spanning maps, are not implemented as fully as we would have liked.

In the short term, the buildtools are currently under review and are likely to be superseded by a more modular and portable replacement. In the medium term, it is likely that the LCFG server will be rewritten with functionality similar to that of the existing server but with a more modular and maintainable structure.

The component library should continue to evolve and grow, including support for new subsystems and platforms.

7.2 The Future

LCFG has provided a very useful platform for studying the application of system configuration in a real-world environment. This has raised a lot of interesting problems that will need more research before we have any good, practical solutions.

The study of configuration languages is one important issue: how do we create a language that allows people at different levels to express their configuration requirements and compose them without conflicts? The issue of centralised vs. decentralised management is also important: can we implement a distributed method of composing configurations that is more efficient and robust than the centralised solution? A desire for more autonomics is another motivating force for fully automated higher-level configuration.

LCFG provides a framework for prototyping new ideas, as well as a valuable source of data on how systems are configured in practice. The publications links on the Web site show some LCFG-related research.

However, existing configuration tools are incompatible. Related areas, such as network configuration and grid application configuration, have their own technologies again, even though they share many of the same problems. Perhaps the future will bring more standardisation and some degree of unification and compatibility.

Appendix A. Bootstrapping an LCFG Installation

The demonstration image used for the tutorials contains a pre-installed version of LCFG. In a full production system, LCFG itself will install new machines complete with all the necessary LCFG software already present. In both of these cases, there is no real need for most users to understand the details of the LCFG installation. However, if you have been using the demonstration image and would now like to deploy LCFG on some real hardware, you will probably want to install the LCFG core manually.

The basic LCFG software will run on a variety of platforms, from Solaris to Mac OS X. But you will probably need to rebuild packages and/or install software manually, unless there are prebuilt packages for your OS on the Web site. We recommend that you follow this manual installation process initially under a version of Linux similar to the one supplied on the tutorial image; this will allow you to familiarise yourself with the process before tackling any portability problems. You will need the following from the LCFG Web site:

- Core packages: RPMs containing the "core" LCFG code: the client, the LCFG server, the libraries, and some basic components.
- Prerequisites: Non-LCFG RPMs containing prerequisite packages from elsewhere—for example, Perl modules used by LCFG that are not part of the standard OS distribution.
- Core defaults: RPMs containing the schema files for the core components.
- Data files: An RPM or tarball containing all of the standard "header files."

Lists of URLs are provided for the package sets so that you can download and install them easily using wget. For example:

```
$ mkdir download
$ cd download
$ export URL=http://www.lcfg.org/download/rh9/release
$ wget $URL/latest/lcfg-core.urls
...
$ wget -i lcfg-core.urls
...
$ wget $URL/latest/lcfg-core-prereq.urls
...
$ wget -i lcfg-core-prereq.urls
...
$ wget $URL/latest/lcfg-core-defaults.urls
...
```

```
$ wget -i lcfg-core-defaults.urls
...
$ rpm -i .rpm
...
```

This installs, in /usr/lib/lcfg/components, LCFG components, as well as various utilities and libraries. Installation of the LCFG server and client can be verified by checking the usage:

```
$ mkxprof -V
++ warning: no persistent state ...
++   (use -c option ...
usage: mkxprof [opts] [file ...]
...
$ rdxprof -V
usage: rdxprof [opts] [host]
...
```

You should now be able to bootstrap the LCFG client and server:

- Create a profile for the local machine which includes (at least) the client and server components. You can create this using the LCFG server on the demonstration disk (probably easiest), or by running mkxprof from the command line (but this will require the header and schema files in the correct places).
- Run rdxprof at the command line to read this profile (you will need to supply the appropriate options).
- You should now have client and server resources available. Check these with qxprof.
- You should be able to start the client and server components with om.
- You will need to configure the Web server to export the profiles. Look at the Apache configuration on the tutorial disk.
- Copy the source file for your machine into the source directory. The LCFG server should notice the appearance of this file and compile it locally.

This is only an outline description of the bootstrap process: at this stage, it is assumed that you are interested in investigating LCFG a little more deeply, and you should be prepared for some experimentation.

Once you are comfortable with the manual installation process under Linux, you should be in a position to experiment with other platforms if required. If there are no prebuilt packages for your platform, the source can be downloaded from the CVS links on the Web site.

Appendix B. Buildtools

LCFG does not enforce any special process for building components, but most people currently use a common set of Makefiles and scripts known as the *buildtools*. These have evolved over the years, rather than being designed, and they have a number of issues—particularly with portability and modularity. In particular, the buildtools assume the use of CVS for version control and work best when using Red Hat RPMs for packaging.

It is likely that the buildtools will be superseded at some point, but for now, they provide a very convenient set of facilities:

- Substituting build-time configuration variables into scripts, TeX documents, and other files.
- Automatically incrementing version numbers and committing new releases with the appropriate tags.
- Automatically building RPMs, Solaris packages, or Mac OS X packages, all from specific CVS versions or the working copy.

The necessary Makefiles and scripts are available in the lcfg-buildtools package.

Getting Started

The easiest way to get started with the buildtools is to use lcfg-skeleton to create a set of template files for a new component.

config.mk is the main configuration file—this defines the build-time configuration variables for the package. You can define your own variables, but the skeleton will populate the file with some common ones:

```
NAME=lcfg-module-name
DESCR=description
V=version
R=release
GROUP=LCFG/Components (for example)
AUTHOR=name <mail>
DATE=dd/mm/yy hh:mm:ss
```

The component Makefile should include buildtools.mk close to the start of the file (but following the declaration of any default target):

```
include buildtools.mk
```

buildtools.mk includes the config.mk file, as well as lcfg.mk, os.mk, and site.mk, which provide LCFG-, OS-, and site-specific configuration variables. You will need to provide

a site.mk file for your own site; a sample one is provided on the tutorial image. The file test.mk defines values to be used when testing (see section 6.3, above).

All configuration variables defined in the files mentioned above are available for use in the Makefile. These variables can also be substituted into other files at build-time.

Substitution

buildtools.mk provides the target config.sh, which creates a script to substitute strings of the form @VAR@, with the value of the variable VAR, for all configuration variables.

A generic rule is supplied to create any file foo automatically from the file foo.cin by generating and applying config.sh. The component author normally creates .cin files, and the corresponding target files are configured and generated when they are referenced by the Makefile.

The target config.tex creates a file of TeX definitions for all the configuration variables. This can be included in TeX documents using:

```
\input{config.tex}
```

The TeX variables are named \cfg*name*, where name is the lowercase version of the variable name.

Creating New Releases

The following targets edit config.mk to increment the appropriate component of the version number (X.Y.Z) and then commit all files into CVS and tag them with the new version tag:

- release: Bump the Z component (not the RPM release).
- minorversion: Bump the Y component.
- majorversion: Bump the X component.

A record is also added to the ChangeLogfile (which must exist) to indicate the new release, and the DATE variable in config.mk is automatically updated.

Creating Distribution Tar Files

The target pack creates a tar file from the version of the software in the CVS repository corresponding to the version number in the current config.mk. Apart from config.mk, the working files in the current directory are not used. The tar file is created in the standard RPM source directory.

Other versions can be packaged as well:

```
make V=some-version pack
```

The target devpack creates a development version of the tar file from the files in the working directory. (This might not produce correct results if files have been removed or added since creating the last release.)

The Makefile may define a prep or devprep target, which is called immediately before packing the files into a tar archive and can be used to delete or manipulate files before packaging. The files are copied to a temporary directory before packing, so any changes here will not affect the working directory or the CVS contents. These targets should be followed by a double colon, since default (null) targets are included in buildtools.mk.

Creating RPMs

lcfg-skeleton provides a skeleton *specfile*. This is the file used by RPM to define the package contents. The buildtools provide rpm and devrpm targets, which pack the appropriate sources, create a working specfile by substituting any variables in specfile, and build the RPMs.

The targets spec and devspec will pack the sources and create the specfile without continuing to build the RPM. This is useful if the RPM is to be built on a different platform. The target devinst builds a development RPM and installs it on the current machine (this requires that the nsu command is available and provides the user with sufficient privileges to perform the installation).

The variable TARFILE is set to the name of the source tar file and should be used in the specfile. The ChangeLog entry for the specfile is automatically created from the ChangeLogfile.

The variables PROD and DEV can be used to prefix specfile lines that should appear only in the production or only in the development version of the RPM. These variables are set to # or null as appropriate.

When creating development tar files and RPMs, the RPM release number will be incremented for each new generation. This provides a way to distinguish between different versions, which may be generated rapidly during development and testing (these RPMs are never released).

Creating Solaris and Mac OS X Packages

The targets pkg and devpkg can be used under Solaris to build Solaris packages instead of Linux RPMs. The Solaris package is created automatically from the information in the specfile by the pkgbuild program. This conversion is not perfect—for example, dependency information is not converted, care is needed with any pre- or post-scripts, and only simple specfile directives are processed. However, it is sufficient for many cases.

The environment variable $PKG_BUILD_DIR can be used to specify the location of the resulting packages.

Mac OS X packages can be created with the targets osxpkg and devosxpkg.

Rebuilding RPMs

Copies of buildtools.mk, os.mk, site.mk, and lcfg.mk are automatically included with the SRPM and are used during rebuilding. This prevents errors if the installed version of these files does not match the version used when the module was packaged (or if they do not even exist).

Any operation that requires software which may not be present on a foreign target system should be performed at build-time rather than at RPM rebuild time, if possible. For example, modules that require specific LaTeX packages to build the documentation can create the PDF file at packaging time using the prep target. RPMs can then be rebuilt without rebuilding the documentation.

Miscellaneous Targets

- Any clean target supplied by the component Makefile should be followed by a double colon, since buildtools.mk provides a default target to remove common files.
- A generic rule is provided to create lcfg-foo.$(MANSECT) or foo.$(MANSECT) from foo.pod.
- Adding the following rule will cause make release to fail if there are files in the working copy that are out of date with respect to the repository:

  ```
  uptodate:: checkcommitted
  ```

- Adding the following rule will force the ChangeLogfile to be generated from the repository contents:

  ```
  changelog:: cvschangelog
  ```

Branches

Branches can be created as follows:

```
cvs tag -b branch_module_X_Y_Z_branch
cvs update -r branch_module_X_Y_Z_branch
```

Edit the config.mk to include:

```
BRANCH=_branch
```

Environment Variables

A number of environment variables can be set to change the behaviour of the buildtools.mk makefile:

$REL_PFX: The value of this environment variable is added as a prefix to RPM release numbers. This can be used to indicate the environment/site in which the RPMs were built (this may involve, for example, different versions of various libraries).

$INC_DIR: The location of lcfg.mk, site.mk, os.mk, and buildtools.mk if they are not in the standard /usr/include location.

$CVS_PFX: The prefix used when accessing CVS modules. This is necessary if the modules are not located in the root directory of the CVS repository.

$PKG_BUILD_DIR: The default is /var/tmp/pkgbuild.

Appendix C. The Linux Installroot

The mechanism used to install machines from bare metal depends heavily on the OS. This appendix outlines the process for Linux—the LCFG *installroot* process. This is described in more detail on the LCFG wiki. A similar process has been implemented for Solaris, based on Jumpstart.

The following sequence is used to perform a bare-metal install:

- The machine is booted from removable media or from the network, using a temporary root filesystem—the installroot.
- The installroot boot process fetches the profile for the node and calls a number of components that are specifically concerned with install-time functions—for example, partitioning the local disk and creating initial configuration files.
- A number of components run to configure various aspects of the local disk. In particular, the `updaterpms` component is run to install the software onto the new system. Apart from the fact that the target filesystem is not the current root, these components work in exactly the same way as they would when reconfiguring a normal running system.
- The node is rebooted on the newly created filesystem, and the installation process is completed by the standard components, which are started as part of the normal boot sequence.

The installation process may require slight modifications for individual sites—for example, there may be differences in the parameters supplied by the DHCP server, or other small differences in site services. The `install` component accepts some configuration to accommodate these differences.

Creating the Installroot

A bootable ISO image of the installroot is available from the Web site, so creation of a new installroot is necessary only if, for example, additional drivers are required.

The installroot is a bootable Linux filesystem. The buildinstallroot program allows this to be created easily from a standard LCFG profile specifying the packages it should contain:

- Create a source file (say, myroot) for the installroot. A suitable default copy is available from the Web site.
- Compile this into an XML profile, exactly as if it were a normal machine.
- Use buildinstallroot to create the installroot image:

  ```
  $ /usr/sbin/buildinstallroot -f -p myroot -o /r.iso
  ```
- This will create an installroot filesystem in /r and an ISO image in /r.iso.

The ISO installroot image can be used to create a bootable CD, which is the easiest way of performing a new installation. The filesystem image of the installroot can also be used to perform a network install using PXE.

Install Parameters

When the installroot boots, it attempts to use DHCP to obtain the network parameters. If DHCP is not available, these parameters can be supplied by providing a file on an (ext2-formatted) floppy disk.

The installroot also needs to know the URL of the LCFG server. This can be supplied by using the DHCP user-class option. A typical DHCP server configuration might include:

```
subnet ... {
  ...
  option user-class "http://server.domain/profiles";
  ...
}
```

If this DHCP option is not present, the URL can be given by specifying a variable in the floppy-disk configuration file. If this is not available, the user will be prompted for the URL of the LCFG server.

Install-time Components

Most of the components that run from the installroot are exactly the same components that will run on the final live system. Some of these components have a specific install method to perform special operations during installation. For example, the client component needs to fetch an initial version of the profile before any of the normal resources are available. The fstab component is responsible for partitioning the local disks according to the resources in the profile (only possible at install).

The install component is the install-time equivalent of the boot component; it determines all of the other commands to be run at install time. In addition to the install methods of LCFG components, these can include arbitrary shell commands, specified as LCFG resources (install.methods). This allows the complete installation process to be specified exactly via the profile. For example, if the DHCP server does not supply a valid NTP server, we can hardwire the NTP server that is used to set the clock at install time, by replacing the command:

```
!install.imethod_gettime \
  mSET(%gettime% ntpdate my-ntpserver)
```

Or we can execute some command before setting the time, by adding another command immediately before this one:

```
!install.imethods      mREPLACE(gettime,mycmd gettime)
!install.imethod_mcmd  mSET(%oneshot% my-command)
```

Index

.cin file, 14
$CVS_PFX, 81
$INC_DIR, 81
$PKG_BUILD_DIR, 81
$REL_PFX, 81
$_COMP, 65
$_DEBUG, 65
$_DUMMY, 65
$_LOCKDIR, 65
$_LOGFILE, 64, 65
$_NOSTRICT, 65
$_OKMSG, 65
$_QUIET, 65
$_ROTATEDIR, 66
$_RUNFILE, 66
$_STATUSFILE, 66
$_TIMEOUT, 65
$_VERBOSE, 65

access control, 30
acknowledgement, 10
aspect, 4
authorisation, 30

bare metal install, 47
boot component, 37
bootstrapping LCFG, 76
buildinstallroot, 82
buildtools, 50, 78

C preprocessor, 21
cfengine, 2
chatterd, 13
ClearPwrCycle(), 62
ClearReboot(), 62
client, 4
client component, 37
clone, 47
cluster, 25
comments, 21
component, 4, 34
    method, 35
    output, 62
components
    boot, 37
    client, 37
    example, 50
    file, 10, 38
    inv, 39
    inventory, 39
    logserver, 40
    openssh, 12
    perlex, 50
    profile, 37
    rpmcache, 43
    updaterpms, 19, 43
conditionals, 21
configure method, 35
context, 27
core software bundle, 7

daemons, 68
Debug(), 63
declarative, 5
defaults file, 14, 19, 50
devinst, 80
devpack, 79
devpkg, 80
devrpm, 80
Dispatch(), 58, 59
Do(), 62
dotdef file, 19

early reference, 24
EndProgress(), 63
Error(), 63
example component, 50

Fail(), 63
file component, 10, 38

generic components, 58
genhdfile, 43

header file, 18, 46

Info(), 63
installation, 47
installroot, 82
inv component, 39
inventory component, 39
IsStarted(), 62

language, 50
late reference, 24
lcfg-ngeneric, 58, 59
lcfg-utils, 57, 62
LCFG::Authorize, 36
LCFG::Component, 59, 60
LCFG::Resources, 57
LCFG::Template, 58, 60
LCFG::Utils, 57
lcfglock, 66
lcfgmsg, 57
liblcfgutils, 57
LoadProfile(), 62
LoadStatus(), 62
Lock(), 62, 66
locking, 66
logfile, 64
logfile rotation, 64
LogMessage(), 63
logrotate, 64
logrotate method, 35
logserver component, 40

macro, 21
majorversion, 79
managed components, 39
manual pages, 20
method
  configure, 35
  logrotate, 35
  monitor, 35
  reset, 35
  restart, 35
  resume, 35
  run, 35
  start, 35
  status, 35
  stop, 35
  suspend, 35
  unlock, 35
minorversion, 79
mkxprof, 29
monitor method, 35
mSET(), 22
mutation, 17, 22

ngeneric, 58, 59

OK(), 63
om command, 36
openssh component, 12
optional software bundle, 7

pack, 79
package list, 19, 25, 41
perlex component, 50
pkg, 80
pkgbuild, 80
prep, 80
preprocessor, 21
profile, 4
profile component, 37
Progress(), 63
pseudo nodes, 57
publish, 24, 54

quotation characters, 23
qxprof, 10, 57

reconfiguration order, 67
references, 24
release, 79
repository, 43
RequestReboot(), 62
reset method, 35
resource list, 21, 52
resource type, 51
resources, 4, 20
restart method, 35
resume method, 35
rpm, 80
rpmcache component, 43
rpmcfg file, 29, 41
run method, 35

SaveStatus(), 62
schema file, 14, 19, 50

schema version, 19
security, 30
server, 4, 17
  access control, 30
  authorisation, 30
  security, 30
SetPwrCycle(), 62
shell components, 58
skeleton, 14
skeleton component, 57
software updating, 41
sorting, 53
source files, 4, 18
spanning map, 5, 17, 24, 54
specfile, 80
standard software bundle, 7
start method, 35
StartProgress(), 63
status display, 31
status method, 35
status page, 10
stop method, 35
subscribe, 24, 55
suspend method, 35
sxprof, 58, 60
system configuration, 1

tag list, 21, 52
template processor, 60
test.mk, 14, 70
testing, 70
type, 51

unlock method, 35
Unlock(), 62, 66
updaterpms, 19
updaterpms component, 43
utility functions, 62
utils, 57

validation, 51

Warn(), 63

About the Author

Paul Anderson (http://www.homepages.inf.ed.ac.uk/dcspaul/) is the original author of LCFG. He has worked in system administration for over 20 years, both as a practitioner and as a researcher. Paul currently works for the School of Informatics at Edinburgh University, was programme chair for the LISA 2007 Conference, and is the author of the SAGE booklet *System Configuration*.