The Complete Guide to

LCFG

Revision 0.99.63

06/01/05 14:10

Contents

Chapter 1

Introduction

This guide is an attempt to provide a single source for all the practical information necessary to understand and use the LCFG configuration system. It superceeds several other documents, including “Getting Started with LCFG” and “Writing LCFG Components”. The published papers [AS02, AS00, And00, And94] still provide a good general overview, although the older papers are somewhat dated and the detailed implementation descriptions are no longer valid.

This guide includes a large quantity of documentation automatically extracted from the software packages. This provides a useful single reference, but since LCFG is still evolving, it is important to check the version of the software package being used, and to refer to the online documentation if this is more recent.

Software distributions and further details are available from http://www.lcfg.org/.

Acknowledgements

Many people have contributed to the development of LCFG, by writing code, providing ideas and feedback, and suffering the consequences of a prototype implementation on a production network. Paul Anderson created the original system and developed the LCFG core. Alastair Scobie ported the system to Linux and developed many of the core components, including the software update subsystem. Alistair Phipps performed the current Solaris port, and many other people have contributed individual components[1]. Davy Virdee tested the tutorial and provided valuable feedback on the documentation.

Paul Anderson <dcspaul@inf.ed.ac.uk>

[1]The reference documentation for each component lists the original authors.

1.1 Background

LCFG[2] is a system for managing the configuration of large numbers of workstations or other *nodes*. The configuration for an entire site is specified declaratively in a central *database*. New nodes can be installed automatically according to their configuration in the database, and running nodes will automatically be reconfigured if their configuration deviates from the specification, either because the specification has changed, or because of some local action on a node[3].

The LCFG architecture was designed to meet a number of fundamental requirements, and we believe that LCFG is still unique in explicitly addressing many of these issues:

- **Declarative configuration specifications** – the configuration specification defines what the configuration should look like, not the procedural steps required to achieve it.
- **Devolved management** – in any large site, many different people are responsible for configuring different *aspects* of the installation. These people should be able to specify their own requirements independently, and the tool should compose these requirements into explicit configurations for the individual nodes.
- **Variety** – in the extreme case, no two nodes may share exactly the same configuration, although they will have many aspects in common. LCFG is capable of managing all types of nodes on a site, from servers through to laptops. Simply managing cluster nodes, or standard desktops, without also managing their serverss is not considered acceptable.
- **Change** – in a large installation, configurations are in a constant state of flux. Both the required specifications and the actual state of the nodes change constantly and the tool must continually attempt to bring the whole site into line with the required specification[4].
- **Complete automation** – the configuration system must be capable of installing all new nodes and maintaining their complete configuration automatically, otherwise the efficiency benefits of automation are lost.
- **Correctness** – the configuration system should be able to configure systems completely from the declarative specifications with no manual intervention. This permits logical reasoning about the configurations and a introduces high degree of confidence in the correctness of the configurations.
- **Modular Development** – for a tool to maintain the ability to completely configure a node, it is important that new applications or subsystems on the node can be quickly incorporated into the configuration system. This means that it must be possible to write new modules easily, quickly, and independently.

[2] Local ConFiGuration system.
[3] This is sometimes called a *convergent* system in the configuration literature.
[4] This is sometimes referred to as *asymptotic configuration*.

- **Disconnected operation** – LCFG can be used on laptops and other remote nodes which have intermittent network connectivity.

LCFG has a proven ability to manage large and diverse sites very effectively - we believe that this demonstrates the validity of the basic principles. The current software release has also been well-tested in a production environment. However, the following issues should be noted by those considering the use of LCFG at other sites:

- LCFG is designed to manage large, complex installations. No tool can provide a “canned” solution to such problems, and system managers need to understand the configuration requirements of their site, understand the operation of LCFG itself, and provide the site-specific “content” to populate the LCFG framework. The advantages for large and/or complex sites are considerable, but an initial investment of effort is required and the learning curve can be steep.
- The current implementation carries a considerable amount of history, and has often evolved from prototype code. Commands and statements are sometimes not as clear as they could be, and much of the code is in need of refactoring to support further developments.
- LCFG was originally developed under Solaris, but the version described in this guide is currently maintained under Redhat 9. Various components have been ported to a number of other operating systems (Max OsX, Debian, back to Solaris), and these variants are mentioned in the guide where appropriate. However, a configuration system has many small dependencies on the underlying operating system, and these other ports are not so well supported.

1.2 The Future

We now expect the current LCFG implementation to remain as a relatively stable production tool. LCFG has provided a considerable amount of experience in the theory and practice of large-scale system configuration, which has fed an active research programme, and we hope that this will eventually lead to a new generation of configuration tools. This is likely to involve:

- A much clearer special-purpose language.
- Explicit support for composition of independent aspects.
- Some support for configurations specifications involving looser constraints, rather than explicit values.
- A more distributed mechanism for compiling and deploying configurations.
- Integral support for an equivalent of the *context* mechanism.
- An architecture with better support for autonomic fault recovery.

The *lssconf* web site provides some pointers to projects, and a mailing list for discussions of these issues:

```
\href{http://homepages.inf.ed.ac.uk/group/lssconf}{http://homepages.inf.e
```

1.3 The LCFG Guide

- Chapter 2 describes the overall LCFG architecture.
- Chapter 3 is a tutorial which introduces the basic installation and use of the LCFG software.
- Chapter 4 describes the process of deploying LCFG to completely manage an entire site.
- Chapter 5 describes how to create and deploy node configuration descriptions.
- Chapter 6 describes the operation of the components that actually implement these descriptions.
- Chapters 7, and 8 describe the software updating and node installation processes.
- Chapter 9 covers the management of an LCFG server.
- Chapters 10 and 11 provide information for writing new LCFG components. The appendices include copies of many relevant manual pages and full code for some example components.
- Chapter 12 covers the differences in the Solaris port of LCFG.

1.3.1 How to use this Guide

Chapter 2 is a short overview, suitable for all readers. Working through the tutorial of chapter 3 will provide a better understanding of the LCFG principles, and is recommended for those with no previous experience.

Chapters 5, 6, 7 and 8 are important references for those wishing to configure nodes at a site with an existing LCFG installation.

Chapters 4 and 9 are important for those considering the deployment of LCFG at a new site, or those responsible for managing an LCFG server.

Chapters 10 and 11 are for those who need to write their own components for the LCFG framework.

1.4 Notation and Terminology

The following symbols are used in this document:

	Information specific to the DICE[5]installation of LCFG.
	A warning – a common source of errors or other unexpected behaviour.
	Advanced or subsidiary information.

`Fixed width text` is used to indicate literal code or the names of programs and files.

Italic text is used when defining a new term. Italic text in code is used to indicate a value supplied by the user, rather than a literal value.

[5]Distributed Informatics Computing Environment. http://www.dice.inf.ed.ac.uk/.

Chapter 2

The LCFG Architecture

The diagram in figure 2.1 shows the overall architecture of the LCFG system:

- A declarative description of the configuration of each node is created in a text file.
- A node description normally contains a small number of node-specific parameters (*resources*), together with pointers to other files describing various *aspects*, such as “web server”, or “student machine”, or “dell gx260”. The aspect files are created and managed independently by the person responsible for the aspect.
- The LCFG compiler (`mkxprof`) compiles these source files into a single XML *profile* for each machine. The profile contains the expanded set of resources, including the values from all the included aspects. Note that many aspects will overlap and the compiler needs to prioritise and merge (*compose*) values for resources which are specified in multiple aspects.
- A standard web server, such as Apache publishes the XML profiles.
- When a profile changes, the server sends a simple notification to the client node which fetches the new profile from the web server using HTTP (possibly over SSL). Clients normally poll for new profiles periodically in case they miss the initial notification.
- A number of *components*[1]. on the client are responsible for taking the declarative resource values from the profile and implementing the specified configuration on the system. This usually involves creating application-specific configuration files from the profile data and possibly manipulating local daemons directly. The components are completely independent and different nodes will run different sets of components.
- Very simple status information from the components is send back to the LCFG server which maintains a basic monitoring facility.

[1]In earlier versions of LCFG, *components* were known as *objects*.

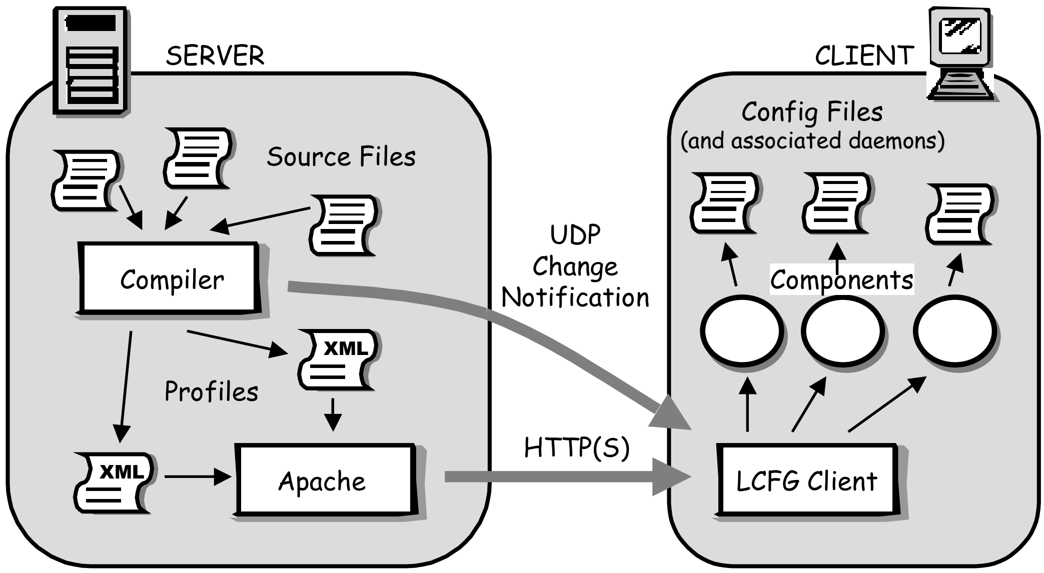

Figure 2.1: LCFG Architecture

2.1 Software Updating and Installation

One component on a node (`updaterpms`) is reponsible for adding and removing software packages on the node to synchronize it with the list of packages and versions given in the configuration specification. In principle, this component could be replaced to manage the software on the node in some other way.

Installation of a new node is treated simply as a special case of the normal maintenance process; a new node is booted from a temporary root filesystem and the normal components (including the software update) are used to synchronize the (empty) disk with the packages and configuration specified in the node description. A small number of components perform specific operations that are only useful at install time, such as disk partitioning.

Notice that the complete site can theoretically be reconstructed from just the repository of packages and the set of LCFG source files.

2.2 The LCFG Software

The LCFG software distribution consists of the following:

- The LCFG compiler `mkxprof`[2] and the component that manages it.

[2] "**Ma**k**e** **X**ml **Prof**ile"

- The LCFG client `rdxprof`[3] and the component that manages it.
- A set of libraries and utilities used by the components. These provide a simple framework and an API for creating components in Shell script or Perl.
- A set of *core components*. LCFG is very modular, and none of the components are completely essential. However, a small number of components are considered as core components because they (or some alternative components implementing the equivalent functionality) are necessary for a useful LCFG installation.
- Optional components. A large number of optional components are available. Some of these can be considered almost as core components, while others will have been written especially for a highly-specific application or environment, and may not be useful outside of their original context.

Appendix J lists the software packages available on the `lcfg.org` web site, which are grouped in convenient "bundles".

[3] "**R**ead **X**ML **Prof**ile"

Chapter 3

Getting Started - a Tutorial

In a production LCFG installation, LCFG itself will install new nodes complete with the necessary LCFG software to maintain their configuration, and it is not essential to understand the details described in this section in order to use the configuration system.

However, this chapter is intended as a step-by-step tutorial to introduce the basic principles of LCFG; it starts by assuming the existence of a node with a pre-installed operating system (but no LCFG), and works throught the following stages. It may be useful to refer to the architecture diagram in figure 2.1 to understand how these steps form part of the overall system.

- Creating a simple node description and compiling this into a profile using the LCFG compiler `mkxprof`. (3.3)
- Using `rdxprof` (the LCFG client) to read and parse this profile. (3.4)
- Running and configuring a single component from the profile. (3.5)
- Publishing the profile using a web server so that a client can fetch the profile from a remote server. (3.6)
- Running `rdxprof` as a daemon (using the `client` component) so that it will automaticaly fetch a new profile and reconfigure components when the configuration changes. (3.7)
- Running `mkxprof` as a daemon (using the `server` component) so that it will automatically recompile source files when they change. (3.8)

Note that the above process is complicated by some bootstrapping issues that are normally avoided when nodes are installed using the LCFG installation process (see chapter 8).

3.1 Prerequisites

This tutorial assumes the availability of at least one node with a copy of Redhat 9 pre-installed[1]. It may be useful to have at least two nodes so that separate machines may be used for the client and the server, although this is not essential and a single node can used for both.

The server node must also support a web server. Basic LCFG operation only involves publishing static XML documents, so any web server should be suitable. However, Apache is the preferred choice and LCFG includes some support for Apache in a production environment; for example, generation of Apache access control files.

⚠ A basic knowledge of Apache (or some other web server) configuration is assumed for this tutorial - the details are not explained here, and it is necessary to understand how to configure the chosen web server to publish documents at a given URL.

3.2 Installing the LCFG RPMs

The LCFG core packages (and their pre-requisites) must be installed. The schema packages containing the default files must also be installed on the server. These are all listed in appendix J, and can be downloaded from `http://www.lcfg.org`.

It is usually easiest to simply install all three of these bundles. For example, to retrieve and install the RPMs for rh9:

```
➜ 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
...
```

Note that some of these (prerequisite) packages may already be installed, or additional

[1] It should be possible to use some other supported operating system, such as Solaris, but this is not recommended, as the there will be a number of small differences which are likely to be confusing.

prerequisite packages may be required. This may generate dependency errors during the installation which must be resolved by deleting or installing the appropriate RPMs.

This installs LCFG components in `/usr/lib/lcfg/components` as well as various utilities and libraries. Installation of the servers 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]
...
```

3.3 Compiling a Profile

The LCFG compiler `mkxprof` takes a *source file* describing a node configuration and compiles it into an XML *profile*. Normally, the profile will include many *resources* for many different *components*, and the source file will specify these, either explicitly, or by including header files describing various *aspects*. This example uses a very simple source file which includes only one component; the Perl example `perlex` (see B.44).

```
profile.components profile perlex
profile.version_profile 3
profile.version_perlex 1

perlex.server foo.bar.com
```

The profile resources specify the components to be included (the profile itself and the `perlex` component), and the schema versions to be used for these components. The `perlex` resource specifies configuration parameters for the `perlex` component (these are documented fully in the manual page – see appendix B.44).

This source file should be created using a text editor, in the current directory. By default, the name of the file should normally be the same as the (short) hostname of the client node for which the profile is being generated. The following examples assume that a single node is being used for both the LCFG client and server, and the short name of this node should be used; we will refer to this as *client*. The filename must not have an extension.

The source file can now be compiled into a profile using the command:

```
mkxprof -S ‘pwd‘ -w ‘pwd‘/WEB -c ‘pwd‘/TMP client
```

The compiler `mkxprof` uses default locations for the source files, the profile output, and the temporary files. Since the default locations are root-owned, the above options are necessary to specify different directories when the compiler is not running as root. The directories will be created if they do not already exist. The options are described fully in the `mkxprof` manual page (appendix C.3).

The resulting profile should be generated under the `WEB` directory. The full pathname depends on both the domain name, and the host name of the client. For example:

```
➜ cat WEB/profiles/domain/client/XML/profile.xml
<?xml version="1.0"?>
<profile
  ...
  <components>
   <perlex>
    ...
    <server>foo.bar.com</server>
    ...
   </perlex>
   ...
  </components>
  ...
</profile>
```

Note that the profile contains separate sections for the two components, and that (among other things), the `perlex` section contains the value specified for the `server` resource.

If there are any errors in the compilation process, the profile will not be generated. Apart from mistakes in the source file, the most likely cause of errors is missing packages; the error message “missing default file” usually indicates that one of the packages containing the schema files has not been installed.

3.4 Reading a Profile

Normally, a client node would fetch the XML profile from the server using HTTP (this is described later). However, to demonstrate the operation of the LCFG client (`rdxprof`), it can be run on the same node as the server, and read the profile directly from the filesystem. This operation needs to be performed as root, since `rdxprof` uses a fixed location[2] for the DBM file which will contain the parsed resources:

```
rdxprof -x WEB/profiles/domain/client/XML
```

The resources should now be available to the LCFG components, and these can be inspected using `qxprof`:

[2] `/var/lcfg/conf/profile/dbm`

```
➜ qxprof perlex
ng_statusdisplay=yes
server=foo.bar.com
ng_schema=1
ng_cfdepend=<perlex
schema=1
ng_reconfig=configure
```

This will display all the (non-null) resource values for the `perlex` component[3]. See the `qxprof` manual page (appendix C.4) for more details.

3.5 Running a Component

Once the LCFG client has parsed the resource values from the profile, these values are available for use by the LCFG components. The `perlex` component is a simple example component which creates a configuration file, using parameters from the profile, and runs a daemon; the daemon simply prints a message to the log file every ten seconds. In a production environment, most components would be started and stopped (by the `boot` component) at system startup and shutdown, but the `om` program can be used to do this manually. `om` would normally be configured with authorization parameters to allow specified users to perform various operations, but for now it is necessary to run the `om` commands as root:

```
➜ om perlex start
[OK] perlex: start
```

The configuration file for the Perlex component should contain the specified `server` parameter:

```
➜ cat /var/lcfg/conf/perlex/config
...
server = foo.bar.com
...
```

The log file for the Perlex component should show the daemon starting with this parameter:

```
➜ cat /var/lcfg/log/perlex
19/11/03 10:46:37: >> start
19/11/03 10:46:37:    configuration changed
19/11/03 10:46:37:    daemon started: version 1.1.3 -
19/11/03 10:46:37:    Hello World: server=foo.bar.com
...
```

[3]Note that the order of the displayed resources is insignificant and may vary. The actual resources themselves may even be different for different versions of LCFG, but this is not important for the purposes of this tutorial.

The daemon is running and will append an entry to the logfile every ten seconds. Use `om` to stop the daemon:

```
➜ om perlex stop
[OK] perlex: stop
➜ tail /var/lcfg/log/perlex
...
19/11/03 10:59:17:    Hello World: server=foo.bar.com
19/11/03 10:59:23: >> stop
19/11/03 10:59:23:    daemon stopped:
```

In a production environment, running components will be automatically reconfigured whenever the source files are changed. Later sections describe how to enable this. For now, the these steps can be executed manually:

- ❑ Make sure the `perlex` component is running (`om perlex start`).
- ❑ Edit the source file to change the value of the `server` resource.
- ❑ Recompile the profile using `mkxprof` (3.3).

  ```
  mkxprof -S ‘pwd‘ -w ‘pwd‘/WEB -c ‘pwd‘/TMP client
  ```

- ❑ Re-run the client to process the new profile, specifying the `-n` option. (3.4). This will cause the client to automatically reconfigure any components whose resouces have changed.

  ```
  rdxprof -n -x WEB/profiles/domain/client/XML
  ```

The logfile should show the component being reconfigured to the new values (`.org` replaces `.com`):

```
➜ om perlex stop
[OK] perlex: stop
➜ tail /var/lcfg/log/perlex
...
19/11/03 11:09:39: >> configure
19/11/03 11:09:39:    configuration changed
19/11/03 11:09:39:    daemon reconfigured:
19/11/03 11:09:39:    Hello World: server=foo.bar.org
```

3.6 Publishing a Profile

In all the above examples, the server and client have been running on the same node and passing the profile via the filesystem. In a production environment, the client will fetch the profile from the server using HTTP. `mkxprof` contains code to fetch the profile, but the server relies on an independent web server to publish it.

The web server should be configured to publish the directory specified in the `-w` option of the compile command. The following examples assume that:

- The compiler `mkxprof` is run without a `-w` option, so that the web directory will default to `/var/lcfg/conf/server/web`:

  ```
  mkxprof -S ‘pwd‘ -c ‘pwd‘/TMP client
  ```

  Note that this probably needs to be run as root to permit writing to the web directory.

- The contents of this directory are published as the root of a virtual web server, typically `http://lcfg.`*domain*.

A knowledge of Apache configuration is necessary to enable this, but the Apache configuration would probably include:

```
DocumentRoot  /var/lcfg/conf/server/web
ServerName    lcfg.inf.ed.ac.uk
```

Starting the web server and running `mkxprof` to regenerate the profile should now make it available via http at some URL such as:

```
http://lcfg.domain/profiles/domain/client/XML/profile.xml
```

This can be verified using an ordinary browser.

The client can now be instructed to fetch the profile from the server, rather than using the filesystem:

```
➜ rdxprof -u http://lcfg.domain/profiles
```

Note that the `-u` option is used to specify the root of the hierarchy containing the profiles. The XML profile will be downloaded into the default location, where it can be inspected:

```
/var/lcfg/conf/profile/xml/client.xml
```

`qxprof` can be used to query the parameters.

It is now possible to create profiles on the server for a number of client nodes. `rdxprof` can be run on each client to fetch its own profile and configure its own components. The source files on the server can make use of C preprocessor *header* files to share common configuration parameters. Header files must have an extension of `.h`.

⇒ When header files are being used, it is normally helpful to compile profiles using the `-d` option to `mkxprof`. This adds additional information to the profile showing the location(s) in the source file(s) corresponding to each resource. The `-v` option to `qxprof` can then be used to locate the definitions. The following example shows where the default value for the resource `profile.format` is defined:

```
➜ qxprof -v profile.format
profile.format:
   value=XML
    type=default
  derive= ... /profile-3.def:22
 authors=default
 context=default
```

Note that these values may differ slightly depending on the version of LCFG being used.

Separate client and server nodes may now be used for the following examples, although a single node may still be used if this is more convenient.

3.7 Running a Client Component

When a change is made to the source files, it is necessary to rerun `mkxprof` (on the server) to generate new profiles, and to rerun `rdxprof` (on each node), to fetch and process the new profile. This section describes how to automate the fetching of new profiles, and the next section describes how to automate the compilation.

`rdxprof` is capable of running as a daemon. In this mode it will listen to UDP notifications from the server, and fetch a new profile whenever it receives a notification. In addition, it can poll the server for changes at regular intervals, in case a notification has been missed. It is possible to simply start `rdxprof` manually with the appropriate command-line options, but LCFG provides a component (the `client` component) to manage `rdxprof`) which allows `rdxprof` itself to be configured from the client source file, and managed with `om`. To configure the client component, add the appropriate resources to the source file for the client:

```
profile.components profile perlex client
profile.version_profile 3
profile.version_perlex 1
profile.version_client 2

perlex.server foo.bar.com

client.url http://server.domain/profiles
client.notify yes
```

This adds the client component to the profile list, specifies the URL for fetching new profiles, and automatically reconfigures components when their resource change. The *server* is the name of the LCFG server, which may be the same as the client if only one machine is being used.

We now need to start the `client` component, so that it will run `rdxprof` to fetch the new profile. However, there is a bootstrapping issue here, because, we need the new profile to specify the resources, before we can start the client component! In a production environment, the node installation process will install the initial profile on the node. For now, we can do this manually, either using `rdxprof` as before, or by using the special `installation` method of the `client` component (which simply calls `rdxprof` with the appropriate parameters):

```
➜ om client install
http://server.domain/profiles
```

The client log should show the receipt of the new profile, and qxprof should display the `client` resources:

```
➜ tail /var/lcfg/log/client
...
19/11/03 10:13:19: >> install
   new profile: http://lcfg ...
     last modified Tue Nov 18 12:20:40 2003
➜ qxprof client.url
url=https://server.domain/profiles
```

Once this is successful, the `client` component can be started:

```
➜ om client start
[OK] client: start
➜ tail /var/lcfg/log/client
...
19/11/03 12:12:35: >> start
19/11/03 12:12:46:    starting daemon [7647/732] ...
19/11/03 12:12:46:    warnings: ...
19/11/03 12:12:46:    context check requested
➜ ps ax |grep rdxprof
...
```

From now on, the client will run automatically; when a profile change notification is received, the new profile will be downloaded and any changed components will be reconfigured. Even the `client` component itself can be reconfigured automatically. To enable the notifications and polling, two additional resources need to be added to the source file:

```
profile.notify true
client.poll 10m
```

The `profile.notify` resource tells `mkxprof` to send a change notification to the client whenever the profile changes. The `client.poll` resource tells the `client` to poll for profile changes every 10 minutes (in case the notification is missed).

If `mkxprof` is rerun, then the notification should be sent to the client which will then fetch the new profile and reconfigure the client component. The client component will actually restart `rdxprof` in this case because the change to the `poll` resource involves changing the command-line parameters for `rdxprof`:

```
➜ tail /var/lcfg/log/client
...
02/12/03 09:22:32:    new profile: http://server ...
02/12/03 09:22:32:      last modified ...
02/12/03 09:22:38:    reconfiguring component ...
02/12/03 09:22:38: >> configure
02/12/03 09:22:39:    configuration changed
02/12/03 09:22:39:    configuration changed: restarting
02/12/03 09:22:39: >> restart
02/12/03 09:22:39:      [OK] client: configure
02/12/03 09:22:40:    termination requested
02/12/03 09:22:40:    stopping server
02/12/03 09:22:42:    starting daemon [19298/732] ...
02/12/03 09:22:42:    warnings: ...
02/12/03 09:22:42:    context check requested
[OK] client: restart
```

The client component can be automatically started at boot time by creating an `init` script, or placing a command in `rc.local`. However, in a production system, this is normally performed using the the `boot` component (see section 6.4.3).

3.8 Running a Server Component

In a typical production enviroment, there will be many hundreds of source files, including both files for individual nodes, and header files which are referenced by other source files. When any of these files are changed, it is necessary to run `mkxprof` on all the affected source files to regenerate the profiles.

As with `rdxprof`, `mkxprof` can run as a daemon and poll for changes to the source files. It maintains a database of dependencies, so that when a header file is changed, it can automatically recompile the source files for all the affected nodes. As with the `client` component, LCFG provides a `server` component to manage `mkxprof`.

To run the `server` component, the source file (for the server node) should include the following resources[4]:

```
profile.components ... server
profile.version_server 2
...
server.poll 10s
```

This tells `mkxprof` to poll for changes in the source files every 10 seconds. It is possible to set additional resources specifying the locations of the various source files (see appendix B.52), however the default values are normally suitable when running as root; `mkxprof -V` shows the defaults:

```
➜ mkxprof -V
...
sources: /var/lcfg/conf/server/source
headers: /var/lcfg/conf/server/include, ...
defaults: /usr/lib/lcfg/defaults/server
packages: /var/lcfg/conf/server/packages
validation: /var/lcfg/conf/server/validation
...
```

Source files should be placed in the `sources` directory, and header files in the (first listed) `headers` directory.

We now need to bootstrap the server startup in the same way as the client:

[4]If separate nodes are being used for the client and the server, the “server” must also be an LCFG “client”, so that it can run the `server` component.

- ❑ Make sure that the source files for all nodes[5], and any header files, are in the the correct directories.
- ❑ Make sure that the `client` component is running on all nodes; particularly on the node that is being used as the LCFG server.
- ❑ Run `mkxprof` with the default pathnames (no `-S` option, etc.) for each node.
- ❑ Check that the profiles have been generated in the correct directory[6] and are being published by the web server.
- ❑ Check that the `client` component on all the nodes has downloaded the new profile.

The `server` component can now be started. (As with the `client` component, this would normally be started by the `boot` component at system startup):

```
➜ om server start
[OK] server: start
➜ cat /var/lcfg/log/server
24/11/03 12:01:08: >> start
24/11/03 12:01:09:    starting daemon [24221/733] ...
24/11/03 12:01:09:    - warnings: ...
24/11/03 12:01:09:    - fetches: ...
...
```

`mkxprof` is now monitoring the source files for changes. To test this, edit one of the source files and change some resource (for example, `perlex.server`). Within 10 seconds (the time specified by the `poll` resource, the server log should show the new profile being generated:

```
➜ tail /var/lcfg/log/server
...
24/11/03 12:01:09: processing: client [1/1, pass 1]
24/11/03 12:01:14: 0 error(s), 0 warning(s) (XML ...
...
```

The client will be notified of the change, and the `client` component will download the new profile and reconfigure the component(s) whose resources have changed.

The error messages from the compilations are recorded in the log file. This is not normally very convenient, and `mkxprof` can display this information on a web page, together with other component status information. See section 9.5 for details.

[5] There may be only one node if the same machine is being used as the LCFG server and client.
[6] `/var/lcfg/conf/server/web`

3.9 Summary

The above steps should have created a small cluster of one LCFG server, and one client (possibly the same machine!) which will automatically maintain its configuration in line with the specification in the source files:

- The configuration of all the nodes is specified in the source files, with common parameters contained in header files.
- The server node is running the LCFG `server` component which uses the LCFG compiler `mkxprof` to recompile the source files for all affected nodes, whenever a source file (or header) is changed.
- The compiler generates a profile for each node which is published by the web server.
- The client node(s) (the server is also an LCFG client) are running the LCFG `client` component which uses `rdxprof` to download a new profile and reconfigure all affected components whenever the profile changes.
- The client node is running a simple example component which configures and manages a daemon according to the specification in the source file.
- The `client` and `server` components themselves are also configured and managed according to the specification in the source files.

Note that this process appears complex, since many operations have been performed manually that would happen automatically in a normal environment that was fully managed by LCFG.

3.10 Where Next?

The following list shows some of the areas that require consideration when developing the tutorial cluster into a production configuration environment:

- Components need to be started and stopped automatically. This is handled by the `boot` component, and is platform-specific. See section 6.4.3.
- Components need to be added to manage other services. See appendix B for a list of those available. Components can be written to manage services for which there is no existing component. This described in chapter 10.
- The software packages installed on a node can be managed by LCFG using a component such as `updaterpms`. This is described in chapter 7.
- LCFG can perform initial installation of nodes according to the specification in the source file. This is described in chapter 8.

- Some thought needs to be given to the organisation of the source and header files. General guidance on this, as well as other useful information about managing a new site with LCFG is given in chapter 4.

Chapter 4

Managing a Site with LCFG

Implementing a new LCFG installation involves:

☞ ** TODO **

Chapter 5

Node Configuration

This section describes how to create and deploy node configuration descriptions for LCFG.

5.1 The Configuration Database

The LCFG "database" is a collection of flat files which specify all the configuration information for a complete site. Node configurations are defined by creating and editing these files. It is possible to simply edit the configuration files with a normal text editor, and this is the simplest procedure during experimentation. In a live site however, there is usually a need for revision control, atomicity, remote access, and locking. Different sites may manage these issues in different ways, for example, by using CVS or `rfe`. The physical location of the files also site-dependent.

There are four different types of configuration file:

5.1.1 Source Files

Source Files hold configuration information for nodes. Every node must have a corresponding source file which represents the complete configuration of that node. Source files usually consist of a mixture of references to *header* files containing shared configuration values, and explicit configuration parameters which are unique to the particular node. For example:

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

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

Not all source files correspond to physical nodes. Some other entities also have source files, such as printers, and the inventory which collates inventory information from all the node files and presents it as a single file.

dice DICE uses `rfe` for managing LCFG configuration files. The manual page for `rfe` explains the options in detail, but the following usage is most common:

- To edit the configuration for the node `foo`:

  ➜ `rfe lcfg/foo`

- To create a new configuration for the node `bar`:

  ➜ `rfe -n lcfg/bar`

`rfe` handles remote editing, authentication, locking and revision control. There is no real support for transactions, but more than one file specification can be given on the command line and the changes will be commited with only a small time interval between them.

Source file names should not have any extension.

5.1.2 Default Files

Default files have names with the extension `.def` and are often called *dotdef* files. There is (at least) one default file for each LCFG component, and this holds the default values and type information for the configuration parameters used by that component. These are used to typecheck and provide default values for the resources which are specfied in the source files. There may be several versions of a default file for each component to allow the server to support clients which are running different versions of the component. In this case, the *schema version*[1] is part of the default file name.

Normally, the default files are created by the authors of the corresponding components, and installed on the server from an RPM[2]; They should not normally be edited. However, it is possible to create local variants of a default file by creating and using a copy with a local schema version, for example, to add site-specific validation to a particular resource.

5.1.3 Package Lists

Package lists have an extension of `.rpms` and are known as *rpmcfg* files[3]. These files contain lists of packages which can be referenced from the node source files to specify the software to be installed on each node (see 5.2.8). These tend to be used for groups of related software packages, and a source file will usually declare a mixture of rpmcfg files and additional, individual packages to be installed on the particular node.

[1]The schema version does not correspond to the version of the component code, since it only need to change when the format of the resources is changed in an incompatible way

[2]RPMS containing default files have names of the form *name*`-defaults-s`*schema*.

[3]Normally, the packages are Redhat RPMs, but this is not essential.

> dice When a new source file is created using `rfe`, a template is automatically provided showing the available header files. Simple node configurations can normally be created just by uncommenting the required headers, and deleting the others. One day, there may be a GUI interface to make this process even easier.

The package lists can contain either explicit version numbers or wildcards which refer to the latest version in the repository.

5.1.4 Header Files

Header files have an extension of `.h` and include common sets of configuration parameters which can be included by more than one node source file. This is the primary method of structuring configuration information to allow devolved management of different aspects of the site configuration. Hence it is important that attention is paid to the organization of the header files.

For example, some header files might define parameters corresponding to different OS or hardware configurations, and these files would then be site-independent, and managed by whoever is responsible for the corresponding platform. Other header files might contain information about site policy, and would therefore be site-specific, and managed by a local site manager. See 5.1.1.

5.2 Configuration File Syntax

The syntax of the LCFG source files has evolved considerably since the original implementation. It is well recognised that the current syntax is not at all clear, and badly in need of replacement. However, the basic elements are simple, and the facilities are adequate. New configuration languages are currently an active research area (see section 1.2 and we hope to eventually replace the present language with something much cleaner.

5.2.1 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 Division of Informatics
```

The documentation for the individual components describes the supported attributes. The component may specify constraints on the acceptable values for a resource and these are validated by the compiler. Some common constraints are often referred to as *types* (for example, `integer`) although these are simply syntactic constraints on the acceptable values, rather than a formal type system.

Components are intended to be modular and they do not normally access attributes of other components, although the source file may specify the value of a resource by reference to some other resource (see 5.2.6).

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 5.2.4). If no value is supplied for a resource, then the default value from the component's default file is used.

The `profile` component is a special case. There is no `profile` component on the client node, but these resources are interpreted as directives to the LCFG compiler. In particluar, the resource `profile.components` declares the components which are to appear in the generated profile. Resources for any components not appearing in this list will be silently ignored. The absolute minimal sourcefile necessary to generate a profile is therefore:

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

(Of course, more components must be declared 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 which has incompatible resources.

5.2.2 Resource Lists

It is often necessary for a resource value to specify a list of items, each of which has a number of associated attributes. Historically, a simple convention known as *tag lists* evolved to represent such lists. This convention has become formalized in recent versions of LCFG, although we would almost certainly have chosen a better syntax if developing a new language from scratch! 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 menubar.

`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[4]. 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.

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
```

5.2.3 The C Preprocessor

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

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

Might be used in a source file as follows[5]:

```
#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 in some cases where some character strings are mistakenly interpreted by the preprocessor: comment characters and string quoting are often sources of trouble. The compiler mutation features described in section 5.2.4 provide some help with quoting awkward cases, but use of the C preprocessor is another design choice that we would 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 which generate multiple source lines. The special character “`¢`” is translated into a newline by the compiler, so that multi-line macros can be created as in the following example:

[4]It is possible for tags names to include underscore characters although this can be ambiguous and is deprecated.

[5]Note that the symbol `HOSTNAME` is predefined by the compiler to the name of the current file.

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

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

5.2.4 Mutation

Typically, individual source files (or other header files) may want to override the values provided in one of the included header files. For example, a header file may define the default disk partitions for all machines of a particular hardware type, but some individual nodes may need to define a different partitioning. If the source file simply declares a new value for the resource, the compiler will signal an error because the same resource has been declared more than once. *Mutation* is the name used to describe to the mechanism which the compiler provides for changing previously defined resource values. The prefix "!" on a resource specification indicates that the following expression is to be treated as a mutation expression, rather than a simple value for the resource:

```
!fstab.size_root   mutation expression
```

It is possible to write a mutation expression to perform any arbitrary transformation of a previously defined resource value. For example, it would be possible to write a mutation that added some constant value to the previously declared partition size. However, this can be extremely confusing and it is recommended that the use of mutations is restricted to a small number of predefined macros. These are contained in the header file `mutate.h` (A.1) supplied with the LCFG server, and described in section 5.1. 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.

`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 vaue of the resource.
`mPRECONCAT(`*A*`)` `mPRECONCATQ(`*A*`)`	Prepend the string *A* to the previous vaue 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.[a]

[a] Care is required when using this function because the DNS lookup occurs only at compile time, and subsequent changes to the DNS will not automatically trigger re-evaluation.

Figure 5.1: Standard mutation macros

For example[6]:

```
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
```

⚠ Note that it is not possible to mutate the default values provided in the component default files. These default values are only used as a "last resort" if no other values have been provided. Resources which have not previously been defined will appear as the null string to any mutations.

⟹ Custom mutation macros can easily be created by defining them in a local header file. The mutation expression should be a Perl expression which accepts the previous value of the resource in `$_` and returns the new value of the resource. 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. See the `mutate.h` header file (A.1) for examples.

5.2.5 Contexts

It is often useful to be able to specify a number of slightly different configurations for the same client, to be used in different circumstances. For example:

- The mail relay on a laptop may need to be different according to the ISP that is being used.
- A disconnected laptop should not attempt to contact a remote Kerberos server for authentication at login.
- A student laboratory machine might be made available for use by remote users outside of opening hours, so the authorised user list might be different.

[6]typically, the first group of declarations would be in some header file, and the second group would be in the source file itself (or a different header file).

- ❑ The set of packages to be included at initial install time might be slightly diferent from the packages to be loaded when the client is fully installed.

The LCFG client maintains an arbitrary set of *context variables* which can be set to arbitrary identifiers, using the `context` command[7]. For example:

```
➜ context
dock=home
➜ context stuff=foo
➜ context
stuff=foo
dock=home
➜ context stuff=
➜ context
dock=home
```

The source configuration 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`, then the mail component will use `mail.myisp.com` as the relay, and in all other cases, it will use `mailhub.ed.ac.uk`.

If the context-specific value of a resource needs to be a variation of the context-free value, then this can be achieved using a *early reference* (see 5.2.6). For example, the following specification will add `apache` to the default `boot.services` except when the context `scheme` is set to `home`:

```
boot.services[scheme!=home] <%%boot.services%%>
!boot.services[scheme!=home] mADD(apache)
```

Context changes on the client can be initiated manually, from cron, or by any other program. In the above case, for example, the context command will automatically be issued by the `divine` network component which manages the network schemes on laptops, and by the `EzPPP` dialup program. Some common contexts include[8]:

[7] The context command uses `om` (6.2) to call the client component `context` method, so that access can be controlled with the client `om` resources.

[8] The `net` variable may not be defined at all if the `divine` component is not running. In this case, the node can probably be assumed to be connected to the local network.

`net=none`	There is no network available.
`net=local`	The node is connected to the local (base) network.
`net=remote`	The node is connected to some other network.
`scheme=`*scheme*	The network scheme, as set by the `divine` component, or EzPPP.
`dock=`*dock*	A laptop is inserted in some particular dock (eg. `home`).
`install`	The node is being installed from scratch.
`power=line`	A laptop is using mains power.
`power=battery`	A laptop is using battery power.

⟹ The context processing is implemented by the LCFG client. The invidivual components see only a configuration change, and they do not need to be aware of whether this is due to a source configuration change, or a change in context. It is also possible for additional context-specific resources to be defined locally so that configuration information can be used even where that information is not available on the server; for example, the information allocated by DHCP to a roaming laptop. This may lead to some resources having different values from those declared in the source configuration files. This capability should therefore be used with caution, and the `divine` component is currently the only case where this is used extensively[9]. The local resource definitions created by these programs are stored under `/var/lcfg/conf/profile/context`.

⚠ The current implementation of context handling in LCFG is not good. If any context-specific resource specification matches the current context, then that specification is used, otherwise the context-free specification is used. 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 which match, then the most recently set context takes precedence. Conditionals which depend on multiple context variables require careful construction to ensure that they are always disjoint, and this is best avoided. Contexts are persistent, even across reboots.

The conditional context expressions must appear in square braces immediately after the resource attribute (no space)[10]. The expressions may include the following:

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

Note that some resources are evaluated on the server, rather than the client (for example, the `profile` component, or the `inv` component). It makes no sense to attach context expressions to these resources.

[9]This is also useful for debugging.
[10]One exception is the use of contexts with packages; see 5.2.8.

5.2.6 References

It is sometimes useful for the value of one resource to refer to the value of some other resource. This can be achieved by using a *reference*. For example, to include the physical location of the node in the login banner:

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

The string `<%inv.location%>` is substituted with the value of the `inv.location` resource which is the physical location from the inventory information.

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

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

Sometimes, this is not what is required. In particular, it may be desirable to copy the current value of some other resource, perhaps because we want to perform a mutation on the copy (see the mail relay example in section 22, for example). A *late reference* is notated using a double percent sign and is evaluated as soon as it occurs. For example, the value if `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)
```

Notice that C preprocessor macros can often be used to achieve a similar effect to references, but the use of references is generally preferred.

```
#define LOC my-location
inv.location LOC
kdm.greetstring HOSTNAME (LOC)
```

➠ It is possible to use references (or macros) to provide a common source of information which may be used by several different components. For example, we could simply define a dummy component (say, `common`)[11] which contained some common information. If several components required the same information, then they could reference the common resources. Only the resources of the common component would need setting on a per-node basis.

[11] A default file would need to be created for this component defining the supported resources.

5.2.7 Spanning Maps

References enable one resource to refer to the value of some other resource of the same node. There is no such general mechanism for referencing resource values from other nodes. However, there are some cases when a particular node really needs to know information about another node; for example, a DHCP server may need to know the MAC addresses of its clients. Clearly, the MAC addresses of these clients could be specified explicitly in the source file for the server, but this is not good, since the correspondence between these values and the actual client source files must be maintained manually (possibly duplicating information, and possibly being inconsistent).

Spanning maps provide a mechanism for nodes to *publish* certain resource values, and for these resource values to be made available to other nodes which *subscribe* to the spanning map. In the above example, the DHCP clients would publish their MAC addresses to a spanning map and the server would subscribe to the spanning map to get the list of clients and their MAC addresses.

The component author decides which resources will be published to a spanning map, and the names of the resources that will be used when the component is subscribed to. In general, it is not necessary to be aware of these details; to use the components it is simply necessary to provide a name for the spanning map. This provides the link between the publishers and the subscribers, and the resource name 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 had decided that it will subscribe to the map named in the `cluster` resource an import the list of hosts into the `host` resource, and 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 serve all the DHCP clients which 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 usual to have map names of the form *service*`/`*cluster*; for example: `dhcp/inf1`[12]. Notice that clients can be added to, and removed from the cluster without changes to the server source file.

⟹ It is possible for a node to be both a publisher and a subscriber to the same map. In this case, the compiler may require several passes to perform the final evaluation, and this

[12]There is no special significance to the `/` symbol.

will be detected automatically. A limit is imposed on the number of such recompilations to prevent an infinite loop in the case of circular references. Nodes which 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 wise to avoid situations where every node is a spanning map subscriber.

5.2.8 Package Lists

The LCFG source files specify a list of *packages* to be installed on the node, including:

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

Normally, the packages are given as Redhat RPM specifications which are interpreted by the `updaterpms` component. However the list may be interpreted by any other component on the client, and there is no reason why the list should not be used to represent packages in any format, providing a suitable component is available to manage them.

The package list could be represented using normal resources, however the LCFG server and client handle the package list as a special case to provide 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 may have one of three different forms:

name-*v*-*r*	The named package is added to the package list. If the specification is preceeded by a "`+`", then this replaces any previous specification of the same package with a different version/release. If the specification is preceeded by a "`-`", then 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 generatde 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 the rpmcfg 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 with the last two being specific to an individual node.

⚠ 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[13].

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

The `updaterpms` component supports a number of flags for controlling various options of the RPM installation. 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 "`:`":

```
/* Do not run pre/post install scripts */
profile.packages_noinst foo-3-4:s
```

`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
```

5.2.9 Semantics

The LCFG language has evolved considerably from its initial simple conception. In an attempt to maintain compatibility, the current language contains several historial artifacts that can be rather confusing. The following situations in particular often cause problems:

- The default values for component resources (from the `.def` file) are only applied, at the end of the compilation process, if no value has been provided for a resource by any other source (or header) file. This means that it is not possible to mutate a default value.

[13] They will be installed the first time the that `updaterpms` component runs after the node is installed

- ❑ If a context is specified for a resource assignment, a separate “context-sensitive” copy of the resource is created. This does not inherit any previous value of the “context-free” resource, and subsequent mutations on either copy of the resource do not affect the other copy.
- ❑ Mutations are frequently used to add package specifications to the profile package list. Individual packages may be prefixed with `+` or `-` which are only processed when the final list is expanded. The interaction between these can be confusing, especially if it is also complicated by context-specifications (see the previos item).

5.3 Configuration Deployment

Large installations will normally have LCFG servers configured to propagate configuration changes to the nodes automatically. This usually happens soon after the new source files have been saved. However, some knowledge of this deployment process is useful for debugging, so it is described in this section, together with the manual alternative.

If any file included by a source file is changed, the entire file must be recompiled into a new XML profile. This profile contains all the expanded resources for the individual node. The server notifies the client of the change, and the client then fetches the new profile. Individual components whose resources have changed are called to implement the appropriate reconfiguration.

5.3.1 Compiling the Profile

The program `mkxprof` (see appendix C.3) is the LCFG compiler. This takes a list of source files and compiles them into XML profiles. This can be run by hand, and any compilation errors will be reported to the terminal, together with the offending files and line numbers.

```
➜ 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, the XML profile will be generated in the appropriate directory (use `mkxprof -V` to see the default directories). Normally, this directory will be published using a web server, such as Apache, making the XML available to the client.

Manual compilation is useful for simple testing, but in practice, it may be necessary to supply a large number of options to `mkxprof` defining the local directories to be used. In many cases, the necessary header files may also only exist on some central server, and not on the local workstation.

The LCFG `server` component (see appendix B.52) is used to run `mkxprof` as a daemon. The daemon maintains a database of file dependencies and regularly polls all the

LCFG files. If any file has changed, all the dependent files are automatically recompiled. The server can also generate HTML status pages for each node to display error messages, rather than requiring them to be retrieved from the server log file. See section 36 for a description of these status pages, and access to the server log file.

dice DICE runs an LCFG server which polls continually for file changes. As soon as any file is committed[a] using `rfe`, then all dependent files will be recompiled. Errors will be shown on the web status page for the client.

[a]depending on how many changes the server is processing, there may be a delay of between a few seconds and several minutes.

5.3.2 Profile Transport

When a profile changes, the server sends a simple UDP notification to the client, but does not wait for an acknowledgement. Normally, the client will poll the 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 server. The XML is parsed and the resources are stored in a local database.

Any components whose resouces 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` (see appendix C.4). If the client is running the `logserver` component (see appendix B.34), then the resources can also be inspected remotely (see 9.3).

⟹ The client component attempts to optimise profiles 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 server and re-parsed. The install method can be also provided with an explicit URL as an argument; this forces the client to fetch the profile from a different server.

Chapter 6

Components

The set of component scripts on the client is responsible for maintaining the node configuration according to the resources in the profile. Each component manages a disjoint *subsystem* of the node configuration; for example the `inet` configuration, or the `sendmail` configuration. The `profile.components` resource defines the components which will have resources included in the profile. The `boot.services` resources (6.4.3) defines which components will be started automatically at boot time.

Component scripts are called by the `client` component whenever their resources change, by the LCFG boot subsystem (6.4.3) at system startup and shutdown, and by various other utilities using `om` (6.2). The scripts are passed a *method* argument describing the required operation, in a similar way to System V init scripts. The method may optionally be followed by standard and/or component-specific options (6.3). Most methods are assumed not to be re-entrant and a per-component lock normally blocks method calls if some other method is currently executing.

6.1 Component Methods

The following standard methods are supported for all components. All methods can be called manually using `om` (6.2), and most methods are also called automatically by other parts of the system:

- `configure` – This is the most important method; it is called whenever the component resources are changed. The component updates the configuration files to reflect the new resource values, and notifies any associated daemons. Note that immediate update of configuration changes is not always sensible and the component may decide to defer certain changes; for example, if a user is currently logged on to the console, the `kdm` component will defer updates which involve restarting the daemon until the user had logged out.

- `start` – This is called at boot time to start a component. An error occurs if the component has already been started.

- `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. A warning (but not an error) is issued if the component is not started.
- `run` – This method is typically called from `cron`, or manually, to perform some ad-hoc operation, often depending on the method options. An error occurs if the component has never been configured.
- `logrotate` – This method is conventionally called by a `logrotate` script to notify any daemons that they should release logfiles.
- `suspend` – This method is called when an APM suspend occurs. There is no lock on this method.
- `resume` – This method is called when an APM resume occurs. There is no lock on this method.
- `status` – This method prints the current state of the resources being used by the component. Note that this may not be the same as the resources currently specified in the profile if an update is pending for some reason. Some components may use this method to make other status information available, when a component-specific option is specified. There is no lock on this method.
- `log` – This method prints the logfile for the component. Different logfiles or formats may be produced by some components if a component-specific option is specified. 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 not widely implemented and is ignored unless the component has been configured.
- `reset` – This method clears the error and warning files which are used by the status display to determine the icon indicating the component status.
- `unlock` – This method forces removal of any locks.

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.

6.2 Om

Since components are simple scripts, it is possible to call them just by executing the script and providing the method as an argument. However, calling components directly in this way is strongly discouraged; the `om` utility should be used to execute component methods. This provides access control for non-root users, sets up a standard environment for component execution, and provides transparency in the location of the scripts. It may also perform other functions in the future which would cause direct calls to behave incorrectly. `om` is called as follows[1]:

➜ `om` *component* *method* [*options*]

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 authorization.

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

`om_acl_`*method*
: specifies the authorization token for the method *method*. The exact meaning of this token depends on the specified authorization module.

The default authorization module is `LCFG::Authorize` which allows the permissions to be specified in the LCFG source file as `authorize` resources (see appendix B.7).

dice Under DICE, the module `DICE::Authorize` is used for authorization. This interfaces to the LDAP-based DICE authorization service, and DICE capabilities should be specified for the authorization tokens.

6.3 Method Options

The standard component framework accepts a number of generic options which can be specified following the method name[2]:

`-d` (dummy)
: The component actions are printed but not executed.

`-D` (debug)
: Print debugging information.

[1] Some documents mention `om` support for remote execution. This did exist in previous versions of LCFG and may be re-implemented in the future, but it is not available in the current implementation – it is normally sufficient to use `ssh` to call `om` on the remote node.

[2] Note that invividual component may not always implement these options correctly.

`-n` (no strict)
: Certain warning and error messages are supressed. For example, trying to stop a component which 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. Note that holding down the shift key when a component method starts executing will also enable this option[3]. This is useful at boot time to enable more verbose logging on certain components.

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

```
om divine.start -v -- -C
```

6.4 Some Common Components

The LCFG system is highly modular and different nodes will normally run different subsets of components, depending on the required services. However, a few components are concerned with managing aspects of the LCFG system itself and these (or equivalents) will usually be present on most systems:

6.4.1 The Profile Component

This is the only component which is mandatory in every profile, since the resources are interpreted by the LCFG server (`mkxprof`) and used to determine how to compile the profile. The `profile` component is not a "real" component, in the sense that there is no code for the the client.

The `profile` resources specify such things as the list of components and packages to be included in the profile (and their versions), and the acces controls on the XML profile. Appendix B.45 describes the supported resources.

[3] Not currently implemented under Solaris.

6.4.2 The Client Component

The `client` component manages the `rdxprof` daemon. This 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 (see appendix B.10).

6.4.3 The Boot Component

By default, LCFG does not use the normal System V init process. Instead, the `boot` component determines what to run (and in what order) when the system runlevel changes. This allows the services and their order to be determined from the LCFG resources, rather than fixed files. It also allows services to be started or stopped dynamically as required when the configuration changes. A mixture of LCFG components are traditional System V init scripts can be managed. For example, the following resources could be used 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 prefix `rc_` or `lcfg_` to distinguish the two different types of service.

The `boot` component can also arrange to call component `suspend`/`resume` methods at the appropriate time, and to call component `run` methods from a single `cron` job (normallly nightly). The boot component options are describ more fully in the manual page (see appendix B.9).

** TODO **
How is the boot component hooked in to the initttab?

In a standard LCFG installation, the `lcfginit` script is also called from the `inittab` to clear temporary LCFG files and perform other initialization at the start of the boot process.

6.4.4 The File Component

The file component is a general-purpose component which can be used to easily create and customize configuration files, directories, or links. This can be used 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, from an RPM, and the substituted values 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 node configurations can now control the php engine simply by setting the value of this variable in their source files. Note that no special code is required.

⟹ If several different applications are to be configured using the file component, it is often convenient to assign each application a separate default 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.

Very small templates can even be included in-line in the resources, avoiding the need for a template RPM. For example, the `bluzez.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 applications include the creation of user home directories, and arbitrary links, and the ability to control file attributes The file component is described in the manual page (see appendix B.17).

6.4.5 The Inventory Component

The inventory "component" is really a pair of "pseudo-components":

The `inv` component can be included in the profile of normal nodes, and used to define basic inventory information for the node; see the manual page `lcfg-inv` (B.25) for details of the available fields. This information is published to a spanning map (5.2.7). For example:

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

The `inventory` component (B.26) can be included in a “pseudo-node” (10.4.9) source file to import the information from the spanning map and make the inventory information for all real nodes available in the single profile of the “pseudo-node”. The following example is the complete source file for an inventory pseudo-component:

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

The `XMLInventory` format module, specified above, can be used to publish the inventory profile in a special format which contains only the inventory information; for example:

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

The perl module `LCFG::Inventory` (F.2) can be used to fetch this file from the server and parse the contents. The demonstration programs `minv` and `jfile-inv` use this module to display inventory information, and to create a Palm Pilot inventory database (in `JFile format`) respectively.

Chapter 7

Software Updating

The LCFG configuration system specifies which packages should exist on the node, and it manages the configuration files for these packages. It relies on an external package management system to perform the actual package installation (and delete/update), and to keep track of which packages are installed.

By default, LCFG uses Redhat RPM to manage the packages, and the `updaterpms` program to control the synchronization of installed packages with the LCFG specification. However, a single component (`updaterpm` by default) is called to perform the software update, and this could easily be replaced by some other update mechanism; the requirement is simply that the `run` method synchronizes the software on the system (by adding and deleting packages) so that it matches the specifiction in the LCFG profile. A different process is used, for example, by the Solaris port (see chapter 12).

The update program may notify the LCFG client (by touching the file `/etc/LCFG-RELEASE`) when an update has been successful. This allows the LCFG status page to display a warning for those hosts which have not had sucessful updates for a specified length of time; see the `profile.maxuptime` resource (appendix B.45). 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 server and flagged if it is not as expected.

7.1 The Package List

The LCFG `client` component maintains a list of required packages in the file:

```
/var/lcfg/conf/profile/rpmcfg/nodename
```

This file is updated every time that a new profile is received which contains changes to the package specifications[1]. The software update component (by default, `updaterpms`) reads this list when its `run` method is called to perform the update. There are several possibilities for configuring exactly when the `run` method is called:

[1] It is also updated when a context change takes place which affects the package list.

- ❑ The `client.runupdate` resource may be set to initiate a software update immediately whenever the list changes. In practice, this is likely to be a little disruptive for users, so one of the following methods is normally used ...

- ❑ The update component is added to the `boot.run` resource so that it it called whenever the `boot` component runs. Normally this happens once nightly – from `cron`, as specified by the `cron.run_boot` resource.

- ❑ For laptop users and other cases where the node may only be connected intermittently, the update component may be run manually. Normal users can be permitted to do this by setting the `updaterpms.om_acl_run` resource, for example to a capability for the user, or simply to `<console>` (for any user at the console). See section 6.2 for a description of ACLs.

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 allowable syntax of these expressions, and their evaluation depends on the update program; the manual page for `updaterpms`, for example, describes the allowable format.

⚠ The use of wildcard versions is very convenient during development, since new versions of packages can be easily installed without changes to the profile. Their use is not recommended for production installations however, since it is no longer possible to tell, just from the profile exactly what software is installed on each machine.

The package specification may optionally be followed by an optional architecture, if the required architecture is different from the default (`i386` or `noarch`).

This may optionally be followed by a "`:`" and a number of single-character flags. The meaning of these flags depends on the update program being used; the `updaterpms` flags are described in the manual page.

☞ ** TODO **
We need an updaterpms man page

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

- ❑ `#include` may be present to include local rpm lists.

- ❑ `#ifdef` is used to allow different sets of rpms to be selected. These are not normally used by the update program, but the `rpmcache` component, for example, defines a special symbol, so that it may obtain a list of all packages, regardless of the current context, since it must maintain a cache which is valid in all contexts.

- ❑ `#pragma LCFG derive` gives the location(s) in the LCFG sources which specified the following package (if known).

- `#pragma LCFG context` gives the context in which the following package was specified (if context-specific). Note that the update program does not need to be aware of contexts. If a context change affects the package list, it will be updated, and the update component will be run (if specified).

7.2 Updating RPMs

The `updaterpm` program compares the installed RPMs with the RPMs specified in the package list and installs/updates/deletes RPMs to make the installed packages correspond to the specification.

The `updaterpm.rpmpath` resource specifies a colon-separated list of `repositories` in which to search for the RPMs themselves. These repositories may be local (or network-mounted) directories, or they may be URLs of http-exported directories. Repositories which are exported via http must also contain a file called `rpmlist` which simply lists the RPMs in repository, one per line; for example, this could be generated by the commands:

☞ ** TODO **
I'm confused by this. Should it be colon or comma-separated? What si the name of the resource? Is the man page right? It doesn't say you can have a PATH.

```
➜ cd respository
➜ ls *.rpm >rpmlist
```

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 entrire RPM file itself to extract the 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
```

⚠ It is a common cause of problems for `rpmlist` or hd files to be missing or out of date. It is strongly recommended that repositories are managed using scripts which ensure that these files are maintained automatically.

dice Under DICE the `rpmsubmit` script is used to submit RPMs to the repositories. This ensures that source RPMs are submitted (when available), and that the necessary files are updated.

7.3 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 desirable to ensure that all the necesssary RPMs are available on the local disk before commencing the update.
- ❑ If a node is liable to be disconnected from the network (for example, a laptop), a local cache of RPMs can be used to re-install 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.

☞ ** TODO **
This example needs doing
We need to say something about rpmcache at install time

⇛ The RPM cache component is based on a Perl module which has functions for reading package list files, and downloading RPMs, that may be useful to other programs. This is used, for example, to automatically download the set of component RPMs and build the appendices for this guide.

Chapter 8

Node Installation

The tutorial in chapter 3 describes how to install and run the LCFG core software on top of an existing Redhat 9 installation. It is perfectly possible to use LCFG in this "lightweight" way, simply to manage the configuration of a few components, while using some other technique for installing and managing the base operating system. However, the real advantage of LCFG only become fully apparent when it is used to manage the entire system. The LCFG install process allows new nodes to be created from "bare metal", using only a repository of RPMs and the profile to describe which RPMs are to be loaded, and how they are to be configured:

- ❑ The node 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 which are particularly concerned with install-time functions such as partitioning the local disk and creation of initial configuration files.
- ❑ A number of normal components are 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 function in exactly the same way as they would when reconfiguring a normal running system.
- ❑ The node is rebooted on to the newly creating filesystem, and the installation process is completed by the standard components when they are started as part of the normal boot sequence.

⚠ The installation process is clearly very specific to the individual operating system. For example, the Solaris port uses a completely different mechanism involving `jumpstart` (see section 12). However, the installation process may even 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 (see below) is designed to be highly configurable, so that such differences can be easily accomodated.

8.1 Creating the Installroot

A bootable ISO image of the installroot is available from `lcfg.org`, so creation of a new installroot is only necessary if, for example, additional drivers are required at install time.

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

- ❏ Create a source file (say, `myroot`) for the installroot. A suitable default copy is available from `lcfg.org`. This should include at least the package `lcfgbuildinstallroot`.
- ❏ Compile this into an XML profile, exactly as if it was a normal node.
- ❏ 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`.

☞ ** TODO **
We need to put a default copy of the installroot source on lcfgf.org
We need a man page for buildinstallroot
Where does buildinstallroot get its profile from ?
I don't have lcfg-buildinstallroot in any of my package sets
We need to put the ISO on lcfg.org

8.2 Booting the Installroot

The ISO installroot image can be used to create a bootable CD, which is the easiest way of performing a new installation.

If the hardware supports PXE booting, then the filesystem image of the installroot can be used to perform a nework install:

☞ ** TODO **
How do we do PXE installs?

8.3 Install Parameters

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

☞ ** TODO **
What is the disk file called
What are the parameters

The installroot also needs to know the URL of the profile 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 profile server.

☞ ** TODO **
What is this variable ?

8.4 Install-time Components

Most of the components which run from the installroot, when building a new system, are exactly the same components which run on the final live system. Some of these components, however, have specific `install` methods to perform special operations at install time. 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 another important example (see appendix B.19). This is responsible for partitioning the local disks according to the resources in the profile[1].

The `install` component is the install-time equivalent of the `boot` component (see section 6.4.3); it determines all the other commands which are run at install time. This is highly flexible, since these commands may be arbitrary shell commands, specified as resources. This allows the complete installation process to be specified exactly via the profile. The `install.methods` resources lists a set of tags for the commands, and the commands themselves are specified by the list elements. For example, if the DHCP does not supply a valid NTP server, we can hardwire the NTP server which is used to set the clock at install time, by replacing the command:

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

[1]Note that changes to the disk-partitioning resours are only implemented at install time; disks are not repartitioned “on the fly”!

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)
```

Chapter 9

Managing an LCFG Server

9.1 Configuring a Server

☞ ** TODO **

9.2 Organising Source Files

☞ ** TODO **

9.3 Server Plugins

☞ ** TODO **

9.4 Authorization and Security

The contents of the LCFG profile should be considered public; any truly sensitive information should be encrypted at the application level, since the profile is plainly visible on both the server and the client, and most sites will want to distribute profiles freely inside the local firewall. However, many sites may want to make profiles available across the Internet, for use by portables and remotely managed nodes.

Since profile distribution is not part of LCFG, and is normally handled by an external webserver, profile access control cannot be completely managed by the LCFG server. However, the server does provide support for automatic generation of access control files which can be used by Apache to configure profile access on a per-node basis:

9.4.1 Access Control Files

Apache normally reads a single access control file called `.htaccess` in the directory containing the profile. However, it is often useful to support more than one access control file for use in different situations; for example, different access restrictions may be required when using SSL or plain HTTP. This can be configured into Apache using directives such as the following:

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

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

The LCFG server can create arbitrary access control files, such as those specified in this configuration by defining resources such as the following:

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

9.4.2 Access Control

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
```

This example would restrict plain HTTP access to the node itself and SSL access to any nodes with an IP address of the form `129.215.*.*`. The access control string should conform to the syntax required by the Apache `allow from` directive.

9.4.3 Authorization

In addition to address-based access control, it is possible to specify basic authorization directives. These apply *in addition* to any access control; if the access control directives are not present, or if they deny access, then a username and password can be used to gain access:

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

The `profile.passwd` specification causes the 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). An explicit password file could be specified to make use of some existing authentication mechanism, rather than using the automatically generated file.

The LCFG client will cache any password that is defined in a profile and use this password when making future requests. Typically, a laptop, for example, 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 authorization is required.

Note that, if neither an `acl`, nor a `pwf` resource appear for a particular access control file, then no access control file will be created[1], and the profile will be freely accessible.

9.4.4 Protecting Other Web Files

The server provides a mechanism using the `linkdir` resource for arbitrary directories to be linked to the web space for publication. 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_cgi     ...
server.dst_cgi     ...
...
```

[1] Any existing file will be deleted.

It is usually desirable to provide access control files for these directories as well as the profiles themselves. This is possible using resources such as the following:

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

In this case, any valid user in the password file is permitted access.

9.4.5 Acknowledgements and Notifications

The LCFG server uses simple UDP packets to notify clients when new profiles are available. The client uses a similar mechanism to acknowledge profile changes and to return status information to the server.

The notification packets contain no data and are therefore not authenticated in any way. It is possible that large volumes of faked notifications could cause a denial of service attack, and if this is considered a problem (unlikely), then notifications should not be permitted and the client should be configured to poll regularly for new profiles.

The acknowledgements do contain important status information. If a password is defined, then the acknowledgement packets will be signed (but not encrypted) using the supplied password, and the signature will be checked by the server. This offers some degree of security, but is still technically suceptible to various DOS and replay attacks.

9.5 The Status Display

LCFG is not intended to provide a full monitoring system, however if the server component is being used to run the compiler as a daemon, then it can maintain HTML pages showing basic status information for each node. Normally, CGI scripts are used to generate these pages "on the fly", but they can also be generated statically (see the resources for the server component in B.52).

These pages show information from three sources:

- Static information obtained by the server when compiling the profile for the node. 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 node (for example, 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 (B.34). This is a small web server which makes the LCFG logs, and other detailed information available directly from the node itself via HTTP. If this component is running, the status page will provide links to the appropriate URLs.

The status summary page is normally available at a URL of the form[2]:

`http://lcfg.`*lcfg-server*`/cgi/index.cgi`

An example is shown in figure 9.1. If the CGI scripts are being used, then the page will also include a an option to enter a query string for selecting the displayed nodes.

The "Help" button displays a page showing the meaning of the various icons. The followng points should be noted:

- The client normally sends acknowledgements when polling for a new profile, or whenever an event change occurs (error, etc). A throttle algorithm prevents clients 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 server resources, but there may be a significant delay (20mins, for example) if the server is recompiling a large number of profiles. The main display be also be out of sync with the individual client dipslays during this time.
- Nodes 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, and is based on the sum of the poll times of the client and server components.

[2]The URL will be different if static pages are being used.

- ❑ Error and warning conditions can only be set by calling the `Reset()` method of the offending components (or by rebooting).
- ❑ If client nodes have an `inv` component in the profile (6.4.5), then the server will publish the inventory fields listed in the `inv.display` resource on the status page.

LCFG: nikita.inf.ed.ac.uk **[Help]**
profile server 2.1.64

inf.ed.ac.uk		
green square blue ! yellow ! yellow dot red power switch client017a	XML	15/12/04 07:43:55
green square blue ! red ! green square magenta * client017b	XML	15/12/04 07:43:55
green square red square gray ? gray ? gray ? client017c	XML	
green square green square red ! red dot green square client017d	XML	15/12/04 07:43:55
green square blue ! red ! blue dot magenta * client017e	XML	15/12/04 07:43:55

Last updated: 15/12/04 07:43:57

Figure 9.1: Summary Page

green square blue ! yellow ! yellow dot red power switch **client017a.inf.ed.ac.uk**	up **[Help]**

Inventory Info	
Model:	Type1
Location:	
Serial No:	3456
Allocated:	fred
Manager:	
Owner:	
Os:	

Status
Client version: 2.0.something
XML profile published: 15/12/04 07:43:52
Last acknowledged profile: 19/04/02 15:53:35
Last acknowledgement: 15/12/04 07:43:55
Last known address: localhost (127.0.0.1)
Last booted: 30/07/02 18:08:59
No errors, no warnings

Components	
blue square yellow ! green square apache	[~~warnings~~] [resources] [doc]
green dot green square green square inv	[resources] [doc]
blue square green square green square logserver	[resources] [doc]
green square green square magenta * mailng	[resources] [doc]
green X green square red power switch nfs	[~~log~~] [resources] [doc]
green dot green square green square profile	[resources] [doc]

Last updated: 15/12/04 07:43:57

Figure 9.2: Individual Client Display

logserver @ nikita.inf.ed.ac.uk : log

[run] [pid] [status] [IMAGE] [IMAGE] [IMAGE] [IMAGE] [IMAGE]

none

logserver @ nikita.inf.ed.ac.uk : log

[run] [pid] [status] [IMAGE] [IMAGE] [IMAGE] [IMAGE] [IMAGE]

Figure 9.3: Logserver Display

Chapter 10

Writing Components

Each subsystem on a node which is configured by LCFG requires a component script to read configuration resources from the node profile and generate the appropriate configuration files and daemon options. If the subsystem involves a daemon process, then the component usually controls the lifecycle of the daemon as well (by stopping and starting it); this allows the component to notify (and perhaps restart) the daemon when the configuration changes, and it allows LCFG resources to control which daemons should run on a particular node. Components also obey certain conventions about their output and logging, so that status information from the components is relayed to the server for display on the status page, and the logs are available via the logserver (see B.34).

Writing a new component usually involves the following steps:

- ❑ Consider whether it is necessary to write a new component at all. The `file` component (6.4.4) can handle most cases which involve only the creation of configuration files. If it is necessary to manage a daemon, or perform more complex processing, then a custom component probably will be required.
- ❑ Create a default file with the types and defaults for the resources to be used (10.4).
- ❑ Choose a language (10.1).
- ❑ Use the appropriate framework (10.3) for the language to create a skeleton component. It is often convenient to start by copying a similar component; the `example` (H.1) or `perlex` (H.2) components are minimal skeletons in shell and Perl respectively.
- ❑ Write code for the `configure` method (10.3.11) to create the necessary configuration files from the LCFG resources.
- ❑ If a daemon is involved, write code for the methods to manage the lifecycle of the daemon (10.3.12).
- ❑ Code any other methods (6.1) that may be require special treatment.
- ❑ Install the component on the client, and the default file on the server (10.7).

Chapter 11 describes the tools that are normally used for building and packaging LCFG components. The use of these tools is recommended, and the examples in this document assume their use. The `example` component (H.1) is a simple illustration of the buildtools in use.

10.1 Choosing a Language

The first consideration when writing a new component is probably to decide on the implementation language; an interface to the standard framework is available for shell (`bash`) and Perl components. Writing components completely in any other language is inadvisable, since this would involve duplicating a lot of the functionality of the framework, and would entail an ongoing maintenence as the framework is upgraded. Of course, it is possible for a shell (or Perl) component to call helper application written in any language. To some extent, the choice of language is a personal decision, however the different languages are suited to slightly different applications:

- If a new daemon process is to be written, and can be written in Perl, then a Perl component is highly recommended; the Perl library component provides support for communicating configuration changes to a running daemon, and for reporting messages directly into the LCFG status system. The `vmidi` component (B.62) is a good example of a simple daemon component in Perl.
- If the component is very simple and just creates a few configuration files, then a shell component is probably most appropriate, especially if those configuration files can be generated by the template processor (10.3.3).
- If the component is intended to manage a pre-existing daemon, then a shell component is usually sufficient. The component must 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, then routines from the framework C library can be added to the daemon itself handle status reporting.
- Perl components may be more portable to other platforms, than the rather bash-specific shell code.

10.2 Portability Issues

The current version of the LCFG core (`utils`, `ngeneric`, `client`), and several components, now run on Solaris, as well as Linux. There is also an experimental port to Mac OS X[Har03]. The following guidelines are suggested to aid in writing components which will be portable across platforms:

- In makefiles and scripts, use the OS-specific symbols defined in `os.mk` (E), rather than explicit program names. The Solaris port depends on the GNU versions of several programs which have non-default names when installed on Solaris (e.g. `gmake`).

- Check for the existence of shell commands or alternatives. For example, the `setsid` and `dnsdomainname` commands do not exist under Solaris, but versions are provided with the `lcfg-utils` module.
- Use `autoconf` or something similar to produce portable C code if this is necessary. Writing portable Perl is usually much easier.
- Try to ensure that the first five letters of package names (after the `lcfg-`) do not conflict with other packages; Solaris package names only use the first five characters. Similarly, for subsidiary packages, ensure that the first three letters of the package plus the last two letters of the subsidiary package name do not conflict with any other packages (or subsidiary packages).
- Do not assume that `hostname` returns a fully-qualified domain name.
- Only the following subset of `specfile` directives are handled automatically by `pkgbuild`. If no other significant directives appear in the `specfile`, then Solaris packages can be created automatically:
  - In the header, `Summary`, `Name`, `Version`, `Release`, `Vendor`, `Source`, and `Buildroot`. Anything else will be ignored by `pkgbuild`.
  - In the main section, `%package`, `%prep`, `%build`, `%install`, `%pre`, `%post`, `%preun`, `%postun`, `%files`, and `%clean`. Anything else will be ignored by `pkgbuild`.
  - In the `%prep` section, only `%setup` is supported.
  - In the `%files%` section, `%defattr`, `%attr` (with comma-delimited attributes), and `%doc`[1].

It is also helpful to add a `Platforms` section to the manual page listing the supported platforms.

10.3 The Component Framework

Creation of LCFG components is supported by the packages `lcfg-utils` and `lcfg-ngeneric`.

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

- `lcfgmsg` (C.2) is a command-line utility, and `LCFG::Utils` (F.5) is a Perl module, both based on the C library `liblcfgutils` (G.1). These routines format and route error and log messages, as well as notifying the client component (and ultimately the server) of status changes (10.3.5).

[1]Relative-path `%docs` that are installed under `/usr/share/doc` with RPM are currently not packaged.

- qxprof (C.4) is a command-line utility based on the Perl module LCFG::Resources (F.3). This 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 the profile. These functions are called automatically by the generic components (see below) and it is not usually necessary to call them explicitly from component code.
- sxprof (C.7) is a command-line utility based on the Perl module LCFG::Template (F.4). This takes a flat-text template file and substitutes variable values from LCFG resources. As with qxprof, resource values can be obtained from several sources. For many components, sxprof is sufficient to generate complete configuration files directly from LCFG resources without any additional coding (10.3.3).

lcfg-ngeneric provides *generic* components which act as superclasses for creating component instances. These provide the default semantics for the standard methods, including resource loading, locking, error checking, and standard option processing. They also provide additional utility functions, and a convenient access to the functions in the lcfg-utils library. The Shell generic component (10.3.1) consists of a file of shell functions which can be sourced by a component shell script. The Perl generic component (10.3.2) is a Perl object class which can be subclassed to create a component instance.

10.3.1 Shell Bindings

The ngeneric (B.39) script provides support for components written in Shell script. Components should simply source ngeneric which provides a number of useful shell functions as well as default code for all standard methods.

ngeneric defines a Dispatch() function which should be called with the command-line arguments. This parses the common options and calls the appropriate method. The absolute minimal component script is therefore[2]:

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

This will support all the standard methods and options, perform locking, 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 redefine the function *Foo*() for any methods that they wish to support. The function Method_*Foo*() can also be redefined in special cases, although this is discouraged, because it is likely to change the standard method

[2] Note that ngeneric uses some bash features, and components should normally specify #!/bin/bash explicitly.

semantics. The generic operations include the locking, loading of resources and some error checking. This means that, 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 $*"
  Info "My server resource is $LCFG_foo_server"
  Info "The verbose flag is $_VERBOSE"
}
```

Note:

- The `Info()` function is a standard function for displaying informational messages. Functions such as this should always be used, rather than simply “echoing” messages (which does not work! see 10.3.5).
- 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` (C.4).
- The exact operations performed before calling the user function depend on the method. These are described in detail in the `lcfg-ngeneric` manual page (B.39).
- The standard options are available as environment variables (see 10.3.8).

The `ngeneric` component also includes a number of other utility functions which are described in section 10.3.4. The manual page (B.39) provides futher details on available variables and functions. The source code for `lcfg-ngeneric` is also quite simple to read, and `lcfg-example` (B.16) provides a complete simple example.

10.3.2 Perl Bindings

The Perl module `LCFG::Component` (F.1) provides a superclass which can be inherited to create pure-Perl components. This module provides all the functionality of 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});
}
```

Note:

- The methods, as well as the utility functions are Perl object methods.
- The resource hash contains resource meta-information as well as values. See `LCFG::Template` (F.4) for details of the format.
- The standard options are available as member variables (10.3.8).

The `LCFG::Component` module includes similar utility functions (10.3.4) to `ngeneric`, as well as the I/O handling functions (10.3.5), and some additional routines for supporting LCFG components which are intended to run as daemons (10.3.12).

The `lcfg-perlex` (B.44) component is a simple example of a Perl component.

10.3.3 The Template Processor

The template processor is a very 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 utility is well-worth studying because it can be used to create most configuration files very easily, with no additional code.

The command-line utility `sxprof` (C.7) is based on the Perl module `LCFG::Template` (F.4) so identical template files can be processed either from Perl, or from the shell. Typically, `sxprof` would be called to read a template and substitute the values of LCFG resources, creating a new configuration file. The values of the resources would usually be 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 LCFG lists is supported automatically, so that multiple lines can be generated for list resources such as:

```
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:%><%%>
```

Yeilds:

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

Note that the syntax can appear complex, but this is largely due to the rather obscure delimiters[3] and 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` (ie. `hda1`). The second field of the fstab is set to `<%mnt_<%item%>%>` which evaluates to `<%mnt_hda1%>` and hence `swap`.

Notice that 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.

[3] The delimiters can be changed with command line arguments, but the default is deliberately rather obscure to reduce the change of misinterpreting any characters which are are literal part of the template file.

- ❑ 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 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 (F.4) for details.

⟹ Note that, 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 which 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 which 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 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:%>
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 (for example, comments), the text can be included inside the delimiters `<%{%>` and `<%}%>`. This will prevent changes to this text from causing a return status of 2, and leading to an unnecessary notification of the daemon.

10.3.4 Utility Functions

The following utility functions are provided:

`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()`
: Save resources from the environment to the status file.

`LoadStatus()`
: Load resources from the status file into the environment.

`LoadProfile()`
: Load 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.

10.3.5 Component Output

Component scripts often run at boot time, or other times when error messages may go unnoticed, and 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 log file. Messages should only be generated on stdout when that is the purpose of the method; for example `log`, or `status`.

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

The following functions are provided to support component output:

`OK()`
: This is called automatically by the `ngeneric` script 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 server for indication on the status display and logged in the log file.

`Error()`
: The component should call this function with an error message. The error is notified to the server for indication on the status display and logged in the log file.

`Warn()`
: The component should call this function to print a warning message. The warning is notified to the server for indication on the status display, and logged in the log file.

`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 log file.

`LogMessage()`
: The component should call this function to print a message to the log file.

`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 which is to be followed by a *progress indicator*. The function `Progress()` should be called at intervals to advance the indicator, and the function `EndProgress()` should be called when the operation is complete.

The following example shows the recommended way of handling long error messages, and debugging messages, so that they do not clutter the display. The environment variables for the standard options are described in section 10.3.8. The verbose option can also be enabled by holding down the shift key when the component method is called[4].

```
[ -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 fancy formats used by Redhat during startup. Newlines embedded in arguments are handled correctly. The C library `lcfgutils` (G.1) 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 in the logfile. If a component needs to print to the standard output, or error (for example as part of a `status` or `log` method) then 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.

10.3.6 Handling Logfiles

The generic component defines the variable `$_LOGFILE` to be the name of the standard component log file. 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 timestamped and formatted messages which are usually preferable.

Sometimes a component may require several logfiles for different purposes, and they should be named by adding extensions to the standard log file name; this makes the logfile visible (when permitted) by the `logserver` component (B.34), and allows the logfiles to be easily rotated using the standard logrotation 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

[4] Not currently implemented under Solaris.

`Reset()` method (or a reboot), so that error messages are not removed until they are manually acknowledged.

The generic `Configure()` method creates a `logrotate` (see `man 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 customize 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 *tag*.

If even more control over the log rotation is required, the component may include a custom template in:

```
/usr/lib/lcfg/conf/component/logrotate
```

The standard logrotation 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 re-open their logfiles.

10.3.7 Monitoring

The component framework provides a number of hooks for interfacing an external monitoring system:

If the resource `ng_monitor` is set to a full pathname, then copies of all events (eg. errors) and monitoring information will be sent to the named file. Typically, this file may be a named pipe, allowing a monitoring daemon to collate the information.

If the resource `ng_syslog` is set to the name of a syslog facility, then all monitoring and events will be written to the named facility.

Some events (eg. errors) are generated in response to normal method calls. The `Monitor()` method is intended to be used by the monitoring system to solicit specific monitoring information from a component. The first argument should be an identifier representing the type of monitoring information required, and the component should respond by calling the `Notify()` function with the requested information. For example, the `mailng` component (B.36) supports a `Monitor()` method which reports the existence (or not) of the sendmail daemon process to the monitoring system. The monitoring system would be expected to poll this method at intervals to monitor the status of the daemon.

10.3.8 Option Processing

The generic component parses the standard options (6.3) and makes them available in the following variables:

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

`$_DEBUG (-D)`
: Print debugging information.

`$_NOSTRICT (-n)`
: Certain warning and error messages are supressed. For example, trying to stop a component which 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)`
: 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.

`$_VERBOSE (-v)`
: Additional messages are printed. Note that holding down the shift key 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.

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
  ...
}
```

10.3.9 Standard Variables

The generic components provide a number of other standard variables:

`$_COMP`
: The component name.

`$_LOCKDIR`
: The lock directory name (10.3.10).

`$_LOGFILE`
: The logfile name (10.3.6).

`$_OKMSG`
: The generic components print the message given by this variable on sucessful 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, the `divine` component shows the current scheme when it starts by setting:

```
_OKMSG="$_OKMSG ($scheme)"
```

`$_ROTATEDIR`
: The directory for log rotate files (10.3.6).

`$_RUNFILE`
: The run file. This file is created as a marker to indictae that the component his staretd.

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

10.3.10 Component Locking

By default, the generic component assumes that most methods are not re-entrant and a per-component lock is established which blocks method calls if some other method is currently executing. Section 6.1 lists those methods which are not subject to locking by default.

The functions `Lock()` and `Unlock()` call the program `lcfglock` (C.1) to make and release the locks. User-supplied method code can call these functions to lock custom methods, or methods which do not normally lock by default. By (conditionally) calling `Unlock()` before `Dispatch()` is 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. Careful manipulation of this can be used to create per-method, rather than per-component, locks by using different directories for different methods.

10.3.11 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, then the component should perform whatever operations are necessary for the daemons to recognise the updated configuration.

The example component (B.16) 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 server, or a local change of context (5.2.5). The node may not even be connected to the network at the time the change occurs, and the component should not need to be concerned with the reason for a particular change.

⇒ Note that immediate update of configuration changes is not always practical and the component must decide whether certain changes should be deferred; for example, if a user is currently logged on to the console, the kdm component will defer updates which involve restarting the daemon until the user had logged out. 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 (10.4.7) 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 resources will be set to <*self* so that the component's `configure` method is called whenever it's 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 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.

10.3.12 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 System V "init" files, and the LCFG boot component (6.4.3) will manage the lifecycle for a mixture of init files and LCFG components. However, using an LCFG component to manage a daemon 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()` and `Start()` to restart the daemon whenever the configuration changed.

⟹ In a normal LCFG installation, the boot component controls which init files and which components should be started. In the above example, the boot component would be configured not to start the init file itself, but to start the component instead (which would then start the init file).

If there is no existing init file, or a more complex startup process is required, 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 IO 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, as shown in section (10.3.3). This allows the `Start()` method to simply retrieve them from the `argfile`, as shown above.

⇒ After starting, or stopping a daemon, it is highly recommended to check that the operation has been sucessful before exiting the method. This might involve, for example, a delay loop which 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 `sendmail` process running. The `Stop()` method of the `mailng` component (B.36) is a good example of how to handle this situation correctly.

⇒ Starting daemons correctly and detecting errors is harder because the daemon may fail asynchronously after it has apparently started sucessfully. 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 `Run()` methods, called from `cron` to check the health of the daemon and report or correct any failures.

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 sucessfull call to the `Start()` method, and before a sucessful call the the `Stop()` method. This is the status reported by the `IsStarted()` function. In a simple case, such as that shown above, this would correspond to the daemon being started (unless it had unexpectedly failed). However, consider the case where a running daemon is an optional feature of the component, so that a daemon is only run if a particular resource is set (the `mailng` component (B.36) is an example of this): in this case, the component must be prepared to stop and start the daemon in the `Configure()` method if the enabling resources are changed. The component may now be "started", although the daemon is not running. It is generally worth taking some care to ensure that both the component and the daemon are in the expected state before attempting to perform any operation such as a restart.

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; ie. 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 adding, using the `lcfgutils` C library (G.1).

If the source code cannot be modified, but the daemon reports error messages to a unique syslog facility, then it should be possible to configure the `syslog.conf` file (B.56) to append error messages on that facility directly to the end of the component error log file (10.3.6). In this case, the errors would appear on the status display, but the notification would not be immediate; it would only occur at the regular client heartbeat, or when some other event occured.

10.3.13 Writing Daemons in Perl

The Perl generic component (10.3.2) can be used to create components without daemons, or components which manage external daemons, as described above. However, it also provides support for writing components which are themselves daemons; ie. the component process forks in the `Start()` method to leave a copy running in the background (both of these alternatives are ilustrated in the Perl example `lcfg-perlex` (B.44)). This has the advantage of providing a much tighter coupling between the running daemon and the LCFG framework; for example, 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 automaticlly 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 which allows it adoprt the new values without restarting.

- All the standard utility functions are available to the daemon process so that error reporting and other logging can use the standard functions, and events are reporetd imediately to the server.

10.4 Default Files

Every LCFG component requires a "default" file (5.1.2) file to define the schema for the resources. This provides:

- Information on on the structure of any list resources (10.4.4).
- Validation predicates ("types") for resource values (10.4.2,10.4.3).
- Default values for resources.

The *Dice Guidelines* document describes how the default files should be packaged and where they need to be installed.

10.4.1 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` and no validation is performed when the resource is compiled.

10.4.2 Builtin 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, with several values (eg. `yes` and `no`) being accepted and 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, except when it is qualified with a validator (10.4.3).

Type specification have the form:

```
@name %type
```

`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 complication. Note that this will not automatically be re-validated if the DNS is subsequently changed.
`vHOSTLIST`	A (space-separated) list of valid hostnames.
`vURL`	A URL.

Figure 10.1: Standard validation macros

For example:

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

Note that a default value for every resource must still appear, even if it is null, and it has a type definition.

10.4.3 String Validation

Arbitrary validation code may be specified for `string` resources. For example:

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

The name in brackets is printed as part of an error message if the value does not satisfy the validation predicate. Care is required in creating validation code, since this allows arbitrary code to be executed in the context of the compiler – this is executed inside a Perl "Safe" module, but infinite loops will still block the entire compiler. It is recommended that the macros supplied in the file `validate.h` (A.2) are used whenever possible (see figure 10.1). Some of these macros also make use of internal server functions to provide more complex validation (for example, a hostname which is valid in the DNS).

10.4.4 Lists

The LCFG *list* resource type supports nested lists of records. The notation for describing resource lists is unfortunately rather awkward. This is a consequence of evolution from the simple list markup convention used in the original LCFG implementation. A list declaration is used to enumerate all the resources which belong to the list and locate their (possibly default) values. The list appears in the XML profile as a nested structure and the declaration is used again at the client end to re-serialise the list into key-value pairs so that it can be accessed as shell variables by a shell component.

A list is declared by specifying the field names that appear in the records (each list element is one of these records). For example, the following declaration specifies a list of records where each record contains two fields:

```
@devices  dev_$ perms_$
```

Defaults should also be provided for each field when a list item does not specify values:

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

An instance of this list might then be defined as:

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

The values appearing in the `devices` resource are known as *tags* which act as unique identifiers for the list elements.

Several different conventions have been used for specifying multi-level lists and all of the known conventions are supported. For example, where the second-level resource keys contain only a single tag:

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

Or where the second-level resource keys contain the tags from both levels:

```
@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 ennumerated tags.

10.4.5 List Sorting

Very often, the value of a list resource is not fully specified in a single file; it is built up from declarations spread across several header files, representing different aspects. For example, the list of components which 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[5] other services, such as a web-service, or a database service.

In some cases, such as the 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, then the order depends on the ordering of the header files, and this can be very error prone.

The LCFG compiler provides a mechanism to have the items of a list automatically sorted 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`, before before `e`. `c` will also come after `d`. The order of unconstrained items in the sorted list is not defined[6] , although some attempt is made to observe the order of the original list. Clearly, it is possible to specify contradictory constraints and this will generate a compile-time error.

[5]using “mutation” – see 5.2.4

[6]The order does vary between versions of the Perl compiler.

The resources containing the ordering constraints must be specified in the definition of the list resource. As well as 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 example, because additional components can be added and their ordering constraints can be included in their own default 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 both).

It may occasionally 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 which must come after `a`, and `before_a` which contains the list of items which must come before `a`. If this definition was used with the resource values above, then 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
```

10.4.6 Spanning Maps

Four types of configuration file are involved in the creation of a spanning map (5.2.7); the subscriber and publisher source files (created by the user) and the subscriber and publisher default files (created by the component authors). This is best illustrated by an example:

- The default file for the dchp client component specifies which resources are to be exported:

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

This specifies 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 dhcp server default 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 which 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 specifies only the map 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 which 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 are changed, all nodes which 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. Eg:

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

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-resouces of the list are not automatically exported[7]. Cross-domain spanning maps require unique (short) node names for the publishers because the short names are used as the list tags in imported map.

10.4.7 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 (B.39). These resources are interpreted by various parts of the LCFG system itself, and control logfile rotation (10.3.6), configuration dependencies (10.3.11), monitoring and status behaviour (10.3.7), and some other options.
- The `om` resources are interpreted by `om` (6.2), mainly for authorization.
- Components should also include a `schema` resource specifying the version of the schema which they require.

10.4.8 Extending Existing Schema

Since the default file is passed through the C preprocessor, it is possible to extend existing component schema by including the default files of those components. Overrides and mutation (5.2.4) are available so that the inherited resources can be changed if required. For example, the `ngeneric` resources for logrotation can be extended:

[7] This is a restriction that we would like to remove.

```
!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 additonal validation; the following example creates a local version of the `client` schema which adds additional validation to the server URL:

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

Note that the header files containing the macros for mutation and validation (for example, the `mEXTRA`) should be included explicitly if they are required:

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

10.4.9 Pseudo-Nodes

Sometimes it is useful to create sources files which do not represent "real" nodes. These source files can useful as either publishers or subscribers to spanning maps. For example:

- An `inventory` source file could be used to collate all the inventory information published by the real nodes. By default, the inventory would be available as an XML file, but plugin modules (9.3) could be used to generate this in a different format if required.
- Source files could be created for printers and the information needed by the print servers could be published to a spanning map. The print servers would then subscribe to the spanning map to get the list of printer names and attributes.
- As a combination of both, a pseudo-node could subscribe to the printer information and feed the resources into LDAP using a plugin module.

10.5 Testing Components

LCFG components are simple scripts, and it should be 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:

- The `ngeneric` component uses several logfiles and status files which require root permission for writing. It is also probably not desirable to write to these live files during testing.
- The resources are obtained from the profile of the current host. These resources may not exist in the profile, or it may be necessary to use different values 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 actions requiring system priviledges that would be undesirable on the live system during testing.

LCFG provides support for all these cases:

10.5.1 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 re-definitions for all the system status and log files 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, however the pathnames for all the individual files can be changed by assigning different values to the corresponding buildtoosl variables. The defaults are defined in `lcfg.mk` and listed in appendix E.

The `test.mk` file is normally included in CVS for the module along with the other source files. The buildtools will not package this file for distribution. This means that any attempt to run the component in the working directory will use the test pathnames, but packaged and distributde components will use the live pathnames.

10.5.2 Test-time resource values

If the buildtools variable `@TESRES@` is defined then 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.

10.5.3 Test-time configuration files

Any buildtools variable definitions in `test.mk` will take precedence over definitions in `config.mk` (or 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.

10.5.4 Test-time daemon 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 a live daemon which requires root priviledges.

The `Do()` function (see section 10.3.4) is also useful for testing; priviledged 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, then a debug message will simply be printed instead.

10.5.5 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 build a production RPM. Test RPMs should never by shipped to production systems since the code is not guaranteed to exist in the CVS.

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

10.5.6 Summary

In summary, the following steps are recommended to simplify component testing:

- ❑ Write the component to include `@TESTSHELLV@` or `@TESTPERLV@` as above.
- ❑ Create a file containing resource values to be used during testing. Define `@TESRES@` 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 which require system priviledges, and test the compoent by using the `-d` option.

❑ Use the buildtools `devinst` target to install a test copy on the current live system.

10.6 Packaging Components

Components are normally created and packaged using the `buildtools` (see 11).

10.6.1 Reconfiguring on Component Upgrade

When a component is upgraded, there may be changes to a template, or the component semantics which require a reconfiguration. This is normally achieved 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, then 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; in particular, the use of the `daemon` command as shown above.

10.7 Installing and Using a Component

Assuming that the component code has been created and packaged according to the"DICE and LCFG Software Guidelines"[And01], the following steps are required to install and use a newly created component:

- The component code must be installed on the client. The RPM could simply be installed by hand, but normally the packages will be managed by LCFG. In this case, the RPM should be placed in the repository, and the name of the RPM added to the `profile.packages` resources, usually by including it in the appropriate rpmcfg file.
- The default file must be installed on the server. The standard build process creates a separate RPM for the default file and this should be installed using the appropriate process.
- Any clients using the component should specify the appropriate schema version:

  ```
  profile.version_component   version
  ```

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

  ```
  !profile.components mADD(component)
  ```

  Usually this is included in some header file.
- If the component is to be staretd at boot time, it should be added to the boot list:

  ```
  !boot.services mADD(lcfg_component)
  ```

  Notice the `lcfg_`! Some other boot resources may need setting (6.4.3) to control the order and run levels.

Chapter 11

Buildtools

The module `lcfg-buildtools` provides a set of makefile targets to assist with the building and packaging of LCFG software from the CVS repository. These provide support for:

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

This chapter should be read in conjunction with the document "DICE and LCFG Software Guidelines"[And01] which recommends guidelines for pathnames and packaging of LCFG components.

Appendix I shows the files from the `lcfg-example` module.

11.1 Getting Started

It is suggested that a test module is created in a temporary local CVS directory for initial familiarisation with `lcfg-buildtools`. It may be useful to use a module such as `lcfg-example` as an initial template.

The module should supply a file `config.mk` which defines the module-specific configuration variables for the package, typically including at least the following:

```
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 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. (see the Software Guidelines document).

The module may also supply a `test.mk` file which provides values to override some configuration variables during testing - for example to use library files from the current directory, rather than the installed system location. This file is used when building the package in the current directory, but it is not included or used when the package is exported.

All configuration variables defined in the above files are available for use in the Makefile. These variables can also be substituted into other files at build-time (11.2).

11.2 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 CVS repository should normally contain the `.cin` files, and the corresponding target files will be 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 lower case version of the variable name.

11.3 Creating New Releases

The following targets edit the `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 `ChangeLog` file (which must exist) to indicate the new release, and the `DATE` variable in `config.mk` is automatically updated.

11.4 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 Linux SOURCE directory (determined by querying with `rpm` to take account of personal rpm preferences).

Other versions can be packed by calling:

```
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. These targets 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 `buildtools.mk`.

11.5 Creating RPMS

The module should supply an RPM spec file called `specfile`. The targets `rpm` and `devrpm` will pack the appropriate sources, create a working specfile by substituting any variables in `specfile` using `config.sh`, 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 is the RPM is to be build on a different platform). The target `devinst` builds a development RPM and installs it on the current machine[1].

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 ChangeLog file.

The variables `PROD` and `DEV` can be used to prefix specfile lines which should appear only in the production, or development versions of the RPM, respectively. 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 is not strictly in accordance with the DICE guidelines, but it provides a way to distinguish between the different versions which may be generated rapidly during development and testing. (These RPMs are never released).

11.6 Creating Solaris 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/post scripts, and only simple specfile directives are processed (see section 10.2 for details). It is however, sufficient for many cases.

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

11.7 Rebuilding RPMs

Copies of `buildtools.mk`, `os.mk`, `site.mk` and `lcfg.mk` are automatically included with the SRPM and 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 which requires software that may not be present on a foreign target system may be best performed at build-time, rather than RPM rebuild time, if possible. For example, modules which 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.

[1] this requires that the `nsu` command is available and provides the user with sufficient privileges to perform the installation.

11.8 Miscellaneous Targets

- Any `clean` target supplied by the module 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 a "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 ChangeLog file to be generated from the repository contents:

```
changelog:: cvschangelog
```

11.9 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
```

11.10 Environment Variables

A number of environment variables can be set to change the behaviour of the `buildtools.mk` makefile. These are mainly intended for use at other sites where the environment may be different:

`$REL_PFX`
: The value of this environment variable is added as a prefix to RPM release numbers. This should be used when building RPMs at other sites to indicate the environment 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 temporary directory in which to build Solaris packages. The default is `/var/tmp/pkgbuild`.

Chapter 12

LCFG on Solaris

Although the LCFG core is relatively portable, many aspects of a complete system, such as installation, and software updating are very dependent on the underlying operating system. The current version of the LCFG core, and some standard components, run under Solaris, and there are Solaris-specific alternatives for performing node installation and software updating. However, the Solaris port is not so widely used as the standard Linux distribution, and it is not likely to be so well supported.

12.1 Prerequisites

LCFG requires a number of utilities and Perl modules which are not part of the standard Solaris distribution. Some of these are available as Solaris packages on the Freeware CD, or from the Sun Freeware repository. Others can can be built from CPAN or distributed tarballs using the `cpan2pkg` utility as follows:

➜ `cpan2pkg` *modulename*
or
➜ `cpan2pkg --from-file` *filename*

Copies of Solaris packages for all these prerequisites are available on `lcfg.org`.

The LCFG buildtools[1] provide `pkg` and `devpkg` targets, analagous to `rpm` and `devrpms`, for creating Sun packages. The packages are generated using the `pkgbuild` program which is described in the manual page. Note that it is necessary to use the GNU `gmake` program, and that several non-default directories are required in the `PATH`. For example[2]:

```
➜ export PATH=/usr/sfw/bin:$PATH
➜ export PATH=/opt/sfw/bin:$PATH
➜ export PATH=/usr/ccs/bin:$PATH
➜ export PATH=/usr/perl5/5.6.1/bin:$PATH
➜ gmake pkg
```

[1] Available in the Sun package `LCFGbuild`.
[2] The syntax of this example assumes that the `bash` shell is being used.

Packages can be added and removed manually using the standard Solaris utilities. For example:

```
➜ gunzip LCFGexamp.pkg.gz
➜ pkgadd -d LCFGexamp.pkg LCFGexamp
...
➜ pkgrm LCFGexamp
...
```

12.2 Solaris-specific components

12.3 Package Management

12.4 Booting

12.5 Installation

Installation of new nodes under Solaris is performed using Solaris Jumpstart. Pre- and post-install scripts (see appendix D) for the Jumpstart installation are used to retrieve the LCFG profile from the server and generate the necessary parameters for the installation.

12.5.1 Jumpstart server configuration

Installation of LCFG clients via jumpstart requires a standard jumpstart server with the following:

- Standard Solaris 9 packages in a suitable NFS exported directory, as described in [Mic].
- The Sun Freeware packages in a suitable NFS exported directory.
- The LCFG package repository in a suitable NFS exported directory.
- An NFS-exported "root" filesystem containing a small set of unpacked LCFG packages and their prerequisites. This provides an image to be used by the client node during the Jumpstart procedure, prior to LCFG being installed locally. Installation into the image directory can be accomplished by using the `-R` parameter to `pkgadd`. Figure 12.1 lists the required packages:
- The root directory must also contain the `cpp`, `gunzip` and `tsort` programs. These are not available as part of the core Solaris packages but are required during installation, so they must be copied from the server's filesystem into the bin directory within the image directory.

`CPANdbfil`	DB_File module for Perl
`CPANdiges`	Digest::MD5 module for Perl
`CPANhtmlp`	HTML::HeadParser module for Perl
`CPANlibne`	Net::FTP module for Perl
`CPANlibww`	LWP module for Perl
`CPANmimeb`	MIME::Base64 module for Perl
`CPANuri`	URI module for Perl
`CPANw3csa`	W3C::SAX::Xmlparser module for Perl
`CPANw3cut`	W3C::Util::Basekit module for Perl
`LCFGclien`	LCFG profile client
`LCFGclis2`	Default resources for LCFG profile client
`LCFGngene`	LCFG new generic component
`LCFGnges1`	Default resources for LCFG new generic component
`LCFGupkg`	Updatepkgs program to keep packages up to date
`LCFGutils`	LCFG resources, libraries and utilities

Figure 12.1: Packages required for LCFG image

- An NFS-exported Jumpstart directory, containing the rules file and the start and finish scripts (see [Mic] for details about how these files are used). It also should contain an LCFG setup script to be executed on the client.

Each directory should be NFS exported read-only. If the exports are read-write, the Solaris Jumpstart installation program will overwrite information in the directories, causing subsequent installations to fail.

All packages, other than base Solaris 9 packages in the core required cluster (`SUNWCreq`), may be in either uncompressed file system (directory) format with the package name being the name of the directory, or in gzip compressed datastream (file) format with *packagename*-*version*-*release*`.pkg.gz` being the format of the filename. Base Solaris 9 packages must be in file system format, as installed by `setup_install\server` (see [Mic]).

12.5.2 Node installation

A small amount of additional server configuration is currently required for each node to be installed. It is hoped that in the future, the Jumpstart server itself will be LCFG-managed and these steps will not be necessary:

- `add_install_client` must be used on the Jumpstart server, as described in [Mic] to add the client.

The node profile must be created and must contain `fstab` and `updaterpms` for the Jumpstart to succeed.

At this point, the client can be rebooted from the network:

```
boot net - install
```

Once the kernel is loaded, the custom `start` script is called (see appendix D). This is used to create the node's Jumpstart profile based on its LCFG profile. It performs the following steps:

- The LCFG root image is NFS mounted.
- The machine's LCFG profile is retrieved using `rdxprof`.
- A Jumpstart profile is created. This specifies that the installation will be an initial install (upgrades are not supported), that the system is standalone and that only the core required cluster (`SUNWCreq`) of packages should be installed. Partitioning is set up as specified in the LCFG `fstab` resources (which must be configured).

Jumpstart then performs the partitioning and installs the specified package cluster. The custom `finish` script is then called. This copies an LCFG setup script from the Jumpstart share onto the target system (into `/etc/rc2.d`), to be executed upon next reboot. The Jumpstart portion of the installation is complete at this point, and the system is rebooted.

Upon first reboot, the LCFG setup script is executed. This performs the following steps:

- The LCFG core image is NFS mounted.
- The LCFG, Sun Freeware and Solaris 9 package repositories are NFS mounted.
- An initial set of directories and symbolic links are created to allow LCFG to function.
- The node's LCFG profile is retrieved using `rdxprof`.
- The list of packages that should be installed is taken from the profile and passed to `updatepkgs`. This performs the necessary package additions, removals and upgrades to bring the installed packages in line with those specified in the profile. For this to work successfully, the `updaterpms` component must be configured and the correct packages profile must be specified.
- Steps are taken to ensure the package repositories will still be mounted after rebooting.
- The setup script removes itself.
- The system is restarted.

Following this, the system reboots in an LCFG-managed state, provided the LCFG `boot` component is configured in the profile and has been installed. The `lcfginit` program must also be installed, to clear temporary directories and set a boot timestamp on startup.

Appendix A

Macros

A.1 Mutate.h

```
/*
 * Standard Mutation Macros for LCFG Server
 *
 * Paul Anderson <dcspaul@inf.ed.ac.uk>
 * Version 2.1.64 : 15/12/04 07:43
 *
 *  ** Generated file : do not edit **
 *
 * 1) Macros with names of the form ....Q() expect their argument
 *    to be a quotde string (Perl string syntax). This allows
 *    arguments to be specified which are not normally acceptable
 *    to the C preprocessor (Eg. containing comment characters).
 *
 * 2) Other macros use the literal values of the argument. In this
 *    the value of the argument must be acceptable to the C
 *    preprocessor.
 *
 * 3) Strings of the form ≪ STRING ≫ (note the spaces) are treated
 *    are "strongly" quoted - ie. the STRING may contain any
 *    characters (apart from ≪≫). Use Alt-Gr/Z and Alt-Gr/X to get
 *    the quite characters.
 *
 * 4) The server treats the character ¢ (Alt-Gr/C) as a resource
 *    separator equivalent to a newline. This allows you to
 *    create multi-line macros by ending lines with ¢\
 */

#ifndef _LCFG_MUTATE_H
#define _LCFG_MUTATE_H

/* Override a value */
#define mSET(A) ≪ A ≫
#define mSETQ(A) A

/* Append an item to a list (space-separated) */
#define mEXTRA(A) ($_?"$_ ":"").≪ A ≫
#define mEXTRAQ(A) ($_?"$_ ":"").A

/* True if list contains item $a (really for use in other macros) */
#define _mCONTAINSA eval ’; s/\s+/’

/* Append an item to a list if not already present (space-separated) */
#define mADD(A) my $a=≪ A ≫; (/\b(\Q$a\E)\b/) ? $_ : (mEXTRA(A))
#define mADDQ(A) my $a=A; (/\b(\Q$a\E)\b/) ? $_ : (mEXTRAQ(A))

/* Prepend an item to a list (space-separated) */
#define mPREPEND(A) ≪ A ≫.($_?" $_":"")
#define mPREPENDQ(A) A.($_?" $_":"")

/* Replace an item in a list (space-separated) */
#define mREPLACE(A,B) my($a,$b)=(≪ A ≫,≪ B ≫); s/\b(\Q$a\E)\b/$b/g; $_
#define mREPLACEQ(A,B) my($a,$b)=(A,B); s/\b(\Q$a\E)\b/$b/g; $_

/* Remove an item from a list (space-separated) */
#define mREMOVE(A) my $a=≪ A ≫; eval ’s/\b(\Q$a\E)\b//g; s/\s+/ /g’; $_
#define mREMOVEQ(A) my $a=A; eval ’s/\b(\Q$a\E)\b//g; s/\s+/ /g’; $_

/* Append a string (no separator) */
```

```
#define mCONCAT(A) $_.≪ A ≫
#define mCONCATQ(A) $_.A

/* Prepend a string (no separator) */
#define mPRECONCAT(A) ≪ A ≫.$_
#define mPRECONCATQ(A) A.$_

/* Replace a substring (no separator) */
#define mSUBST(A,B) my($a,$b)=(≪ A ≫,≪ B ≫); s/\Q$a\E/$b/g; $_
#define mSUBSTQ(A,B) my($a,$b)=(A,B); s/\Q$a\E/$b/g; $_

/* Lookup host IP */
#define mHOSTIP(H) $_=≪ H ≫ ; &$_HostIP

#endif
```

A.2 Validate.h

```
/*
 * Standard Validation Macros for LCFG Server
 *
 * Paul Anderson <dcspaul@inf.ed.ac.uk>
 * Version 2.1.64 : 15/12/04 07:43
 *
 *  ** Generated file : do not edit **
 *
 */

#ifndef _LCFG_VALIDATE_H
#define _LCFG_VALIDATE_H

/* A member of a list */
#define vENUM(L) Enum(≪ L ≫)
#define vENUMQ(L) Enum(L)

/* A line of a file */
#define vINFILE(F) InFile(≪ F ≫)
#define vINFILEQ(F) InFile(F)

/* An IP address */
#define vIPADDR /^(\d+)\.(\d+)\.(\d+)\.(\d+)$/ && \
                $1<256 && $2<256 && $3<256 && $4<256

/* A list of IP addresses */
#define vIPADDRLIST !scalar grep { !(vIPADDR); } split(' ',$_);

/* A valid hostname (in the DNS) */
#define vHOSTNAME Hostname()

/* A list of valid hostnames (in the DNS) */
#define vHOSTLIST HostList()

/* A URL */
#define vURL /^http:\/\/([^\/]+)\// && ($_=$1) && Hostname()

#endif
```

Appendix B

List of Components

B.1 alias

LCFG mail alias component

DESCRIPTION

This component manages the sendmail aliases file.

RESOURCES

addr_*tagtag!alias resource*

The mail address (username) corresponding to the given tag. If this is null, the mail address is assumed to be the same as the tag.

aliasfile

The full pathname of the alias file to be managed.

aliases

A space-separated list of alias tags.

alias_*tagtag!alias resource*

The alias corresponding to the given tag.

basefile

The full pathname of a file (in alias file format) containing aliases to include in the output file. All aliases appearing in this file will appear (in order) in the output file, before any aliases specified explicitly as resources. Aliases in the file which also appear in the explict rresources will be replacde with the values from the resources. This resource is optional.

PLATFORMS

Redhat7, Redhat9

AUTHOR

Paul Anderson <dcspaul@inf.ed.ac.uk >

VERSION

1.0.0-1

B.2 amd

LCFG amd component

DESCRIPTION

This object starts the `amd` automounter.

conftmpl

The template `/etc/amd.conf` file.

gvariables

A list of tags each defining a global variable in the `/etc/amd.conf` file.

gvar_*tag*

The global variable entry associated with the tag *tag*.

maplist

A list of amd map tags.

path_*tag*

The filesystem path for the amd map associated with *tag*.

name_*tag*

The amd map name for the amd map associated with *tag*.

type_*tag*

The type of map (eg hesiod, file) for the map *tag*.

mountoptions

The value required for the AMD_MOUNT_OPTS environment variable - used in various hesiod maps. Optional, and probably Edinburgh specific.

AUTHORS

```
Alastair Scobie <ascobie@inf.ed.ac.uk>
```

VERSION

0.100.10-1

B.3 apache

LCFG Apache component

DESCRIPTION

Simple component to start and stop a default installation of Apache.

RESOURCES

config

The config file to start httpd with. If the filename is not an absolute filename, then it is relative to the *serverroot*. The apache default is used it not specified, this is currently **conf/httpd.conf**. This resource is equivalent to httpd **-f** option.

conftmpl

The *config* template file. If this resources is set, then it will pass the template through `sxprof` with all `apache` resources defined, the resulting output will go to the file specified by *config*. If it is null (the default), then the existing **config** is not changed.

serverroot

Sets the initial value for the ServerRoot apache directive. This can then be overridden in the config file. The apache default is used if this is not specified, this is currently **/etc/httpd**. This resource is equivalent to the httpd **-d** option.

startssl

A boolean value specifying whether the secure web server should be started, otherwise just the regular one will be. This has implications if the server requires a pass phrase, as the machine will stop at boot time waiting for the pass phrase if startssl is true and the SSL certificate requires a pass phrase.

METHODS

start

Calls `apachectl start`

stop

Calls `apachectl stop`

restart

As LCFG spec, ie calls stop then start. Not the same as `apachectl restart`.

ctl [*params*]

Calls `apachectl` [*params*]

ERRORS

It is an error if **conftmpl** and **serverroot**/**config** specify the same file, as data loss is likely to occur. Note the the checking isn't foolproof, and file/path names with .. in it will not work.

NOTES

As this component currently just calls the default apachectl, it will only launch httpd if the config file exists and contains no errors.

RedHat have changed apachectl to source the /etc/sysconfig/apache file for extra options. It is this file that this component updates.

AUTHORS

```
Neil Brown <neilb@dcs.ed.ac.uk>
```

VERSION

1.1.7-1

B.4 apm

LCFG apm component

DESCRIPTION

This component starts the APM daemon (apmd).

proxy

The name of the `apmd` proxy script. If there are APM events defined in the resource `apm.events`, this file will be constructed by resources. Otherwise the script will be assumed to be created by other means (eg RPM distribution).

The default is `/etc/apm/apmd_proxy`.

events

A list of APM events to service. Used to create the `apmd` proxy script.

action_*event*

The action to perform for the specified APM event.

daemonopts

Options for the `apmd` daemon.

AUTHORS

```
Alastair Scobie <ascobie@inf.ed.ac.uk>
```

VERSION

0.100.5-1

B.5 arpwatch

Track MAC/IP mappings

DESCRIPTION

This component kicks off the arpwatch daemon to log ARP packets and keep track of the MAC/IP mappings.

RESOURCES

interfaces

Which interfaces should we watch? This resource must be set.

accept_bogons

Are we interested in reporting bogons, or should we just accept them and put them into the database?

directory

Which directory should the data files live in?

runas

Which user should the daemon run as?

sendTo

Whom should the daemon send mail to?

sendAs

Whom should the daemon send mail as?

arpwatch

What's the path to the daemon?

AUTHORS

```
George Ross &lt;gdmr@inf.ed.ac.uk&gt;
```

VERSION

1.99.13-1

B.6 auth

LCFG auth component

DESCRIPTION

This component contructs all the authorization files allowing access to the machine. This includes `/etc/passwd`, `/etc/group`, `/etc/hosts.equiv` and `/root/.rhosts`.

rootpwd

The encrypted root password.

base_passwd

The base file used to populate `/etc/passwd`.

extrapasswd

A list of passwd entries tags to be added to `/etc/passwd`.

pwent_*TAG*

An additional passwd entry.

base_group

The base file used to populate /etc/group.

extragroup

A list of group entries tags to be added to `/etc/group`.

grpent_*TAG*

An additional group entry.

shadow

This resource, if set to `yes`, will convert the passwd file files to the more secure shadow equivalent.

users

A (space-separated) list of users or netgroups to be added to the `/etc/security/access.conf` file.

owner

A (space-separated) list of workstation owners. Valid usernames in this list will be added to the `/etc/security/access.conf` file.

userhalt

If this resource is non-null, password file entries (with no password) will be created for the users `shutdown` and `reboot` with the shells `/usr/bin/usershutdown` and `/usr/bin/userreboot`.

rhosts

A (space separated) list of items to be added to the `/root/.rhosts`

equiv

A (space-separated) list of items to be added to the `hosts.equiv` file.

nsu

A list of tags, each representing one line in the `nsu.conf` file. If this resource is null, the nsu.conf file will not be changed.

nsu_*TAG*

One line of the nsu.conf file. Note that "%" characters in the value of this resource will be translated into "$" before writing to the configuration file. This allows the use of *%(FOO)* to avoid the problems of shell interpretation for *\$(FOO)*.

tmp_mode

If non-null, specifies the chmod protection mask to be applied to /tmp.

var_tmp_mode

If non-null, specifies the chmod protection mask to be applied to /var/tmp.

consolepermclasses

This is a list of console file and device classes to be defined in the /etc/security/console.perms file.

consolepermclass_*tag*

This is the definition for the class associated with *tag*.

consolepermrules

This is a list of rules for the file and device classes defined in consolepermclasses.

consolepermrule_*tag*

This is the definition for the rule associated with *tag*.

accessrules

A list of rules for the /etc/security/access.conf file.

accessrule_*tag*

The definition for the access rule associated with *tag*.

identdconf

Name of the file to be used as the /etc/identd.conf configuration file.

protectdevs

List of devices (eg disks) which should not be added to the /etc/security/console.perms file. Normally set to the same value as the fstab.disks resource. Note that the device entry should be shortform (eg hda rather than /dev/hda).

AUTHORS

Alastair Scobie <ajs@inf.ed.ac.uk>

VERSION

0.100.8-1

B.7 authorize

LCFG basic authorization module for ”om”

DESCRIPTION

The **authorize** resources are used by the **LCFG::Authorize** Perl module. In a default installation, this module controls which users have the capabilities necessary to execute **om** commands on LCFG components. There is no component code for this module.

Note that LCFG::Authorize is a very basic authorization module which is not suitable for large or complex authorization schemes, and it may not be used in all installations. For example, DICE uses the LDAP-based **DICE::Authorize** module instead - this selection is controlled by the component's **ng_authorization** resource which is normally set to the value of **profile.authorize**.

Components allow a user to run a method *foo* if the user has a ”capability” listed in the **om_acl_***foo* resource. By default, this has the value **om/all**, so users with this capability can execute any component method.

The <lcfgcap > command may be used to query capabilities.

RESOURCES

groups

A (space-separated) list of tags representing groups of users.

users_*groupgroup!authorize resource*

A (space-separated) list of usernames for users in the *group*.

caps_*groupgroup!authorize resource*

A (space-separated) list of capabilities to be given to the users in the *group*.

PLATFORMS

Redhat7, Redhat9, Solaris

AUTHOR

Paul Anderson <dcspaul@inf.ed.ac.uk >, Simon Wilkinson <sxw@inf.ed.ac.uk >

VERSION

0.99.5-1

B.8 bluez

LCFG BlueZ Bluetooth component

DESCRIPTION

This component controls the Bluetooth subsystem.

RESOURCES

hcitmpl

Pathname of template file for hci daemon config.

dund

True to enable dund daemon.

dund_args

Command line args to dund.

hcid_name

hcid config parameter (see hcid configuration file template).

hcid_security

hcid config parameter (see hcid configuration file template).

hcid_pairing

hcid config parameter (see hcid configuration file template).

hcid_linkmode

hcid config parameter (see hcid configuration file template).

hcid_linkpolicy

hcid config parameter (see hcid configuration file template).

hcid_auth

hcid config parameter (see hcid configuration file template).

hcid_encrypt

hcid config parameter (see hcid configuration file template).

hcid_iscan

hcid config parameter (see hcid configuration file template).

hcid_pscan

hcid config parameter (see hcid configuration file template).

helper

The pathname of a command which returns the PIN to be supplied to remote devices when a connection is initiated from the local machine. By default, the value is /usr/lib/lcfg/bluez/getpin - this simply returns the same PIN used for inbound connections (set by the **pin** resource). Note that the Redhat standard is **/usr/bin/bluepin** which is intended to request the PIN via an X dialog, but this does not always work.

pand

True to enable pan daemon.

pand_args

Command line args to pan daemon.

pin

Bluetooth PIN - this is the PIN which should be supplied by inbound connection/pairing requests. Note that a different PIN may be required for outbound connections (see the **helper** resource).

ppptmpl

The pathname of the template file for the PPP config file. If this is null, no PPP config file will be created.

ppp_dns

PPP configuration parameter (see PP configuration template).

ppp_netmask

PPP configuration parameter (see PP configuration template).

ppp_local

PPP configuration parameter (see PP configuration template).

ppp_remote

PPP configuration parameter (see PP configuration template).

ppp_idle

PPP configuration parameter (see PP configuration template).

ppp_extras

PPP configuration parameter (see PP configuration template).

ppp_extra_auth

PPP configuration parameter (see PP configuration template).

ppp_extra_def

PPP configuration parameter (see PP configuration template).

ppp_extra_arp

PPP configuration parameter (see PP configuration template).

ppp_extra_ipx

PPP configuration parameter (see PP configuration template).

ppp_extra_route

PPP configuration parameter (see PP configuration template).

rfaddr_*devdev!bluez resource*

The Bluetooth address of the specified device.

rfbind_*devdev!bluez resource*

If this resource is true, the specified device will be bound by rfcomm at startup (default is true).

rfchannel_*devdev!bluez resource*

The Bluetooth channel for the specified device.

rfdescr_*devdev!bluez resource*

The description of the specified device (comment).

rfdevs

A list of device numbers for rfcomm devices. Each device will appear as **/dev/rfcomm***N*, where *N* is the device number.

PLATFORMS

Redhat7, Redhat9

AUTHOR

Paul Anderson <dcspaul@inf.ed.ac.uk >

VERSION

0.99.11-1

B.9 boot

LCFG boot component

DESCRIPTION

The boot component manages which LCFG components and SystemV init scripts are started or stopped when the system moves from one run level to another (eg boot time, shutdown etc).

It has three main functions :-

- to stop and start components and init scripts (eg at boot time)
- to call the **run** method of certain LCFG components (typically from a nightly cron job)
- to call the **suspend** method of certain LCFG components when a system is suspended, and call the **resume** method of those components when the system is resumed.

Both LCFG components and SystemV init scripts are managed; for brevity, this document refers to *services* to refer to the union of these.

Services stop/start

The boot component recalculates which *services* (LCFG components and init scripts) should be started or stopped at one of two events :-

- the **restart** method is called as a result of the system transitioning from one runlevel to another; eg. on boot or shutdown.
- the **configure** method is called as a result of some configuration change.

The resource list **boot.services** is evaluated to determine which services should be running at the target run level. Each service has an associated resource **boot.levels_***service* which is a list of the run levels that this service should be active in.

The component now produces an ordered list of services which shouldn't be running in the target run level and require stopped. Each service has an associated resource **boot.stop_***service* which indicates two things. Firstly it indicates the service's stop priority level. This is similar to the familiar SystemV rc priority levels; services with lower priority levels are stopped before those with higher priority levels. The priority level can also take the value **NO** which indicates that the service should never be stopped at a transition (used for services which are only ever started). The resource also indicates whether the service should be stopped for this event (**restart** and/or **configure**). The majority of services will only specify the **restart** event; ie the service will only be stopped at a runlevel transition. The use of the **configure** event is described later. A default value for **boot.stop_***service* is **0 restart**; this indicates that the priority is **0** and that the component should only be stopped at a runlevel transition. Having produced this list, the component stops the services in priority order.

The component now produces an ordered list of services which aren't already running but should now be in the target run level. Each service has an associated resource **boot.start_***service* which has similar syntax and meaning to the **boot.stop_***service* resource. The only difference is that services with a start priority of **100** or above are not started if a previous service has requested a reboot.

Once all necessary services are stopped or started, the component checks to see if any service has requested a reboot (by use of the standard LCFG **RequestBoot** macro); if so, the component will schedule an immediate reboot. Only services with a **boot.reboot_***service* resource value including the current event type are checked; the default value for this resource is **restart** which indicates that this component should only be able to trigger a reboot at a runlevel transitition.

Run method

The boot component's **run** method is used to call a specified method, usually **run**, of certain LCFG components to perform, typically, some routine maintenance function. It is normally called nightly (via cron). It does not support SystemV scripts.

The resource **boot.run** is the list of components to be called. Each LCFG component on this list has an associated **boot.user**_*component* and **boot.runmethod**_*component* resource. The first resource indicates which userid to use to call the component; the default value is **root**. The second resource indicates which method to call of the component; the default value is **run**.

This method can also be used to call arbitrary shell commands. If an element of the **boot.run** list has an associated resource **boot.type**_*component* of **direct**, the associated **boot.runmethod**_*component* resource specifies a shell command to be executed.

Once all the specified components have been called, the boot component performs the same reboot check as it does when starting/stopping services. Adding the value **run** to a component's **boot.reboot**_*service* resource will allow that component to trigger a reboot.

Suspend/Resume

The boot component's **suspend** method will call the **suspend** method of those LCFG components listed in the resource **boot.suspend**, in the order as specified in the resource. The **resume** method will call the **resume** methods of the same components, but in reverse order.

Configure event - what for ?

Normally new services which are added to the **boot.services** resource are not started until the next runlevel transition (usually boot time). Sometimes, however, it is useful to start a service as soon as it has been added to the **boot.services** resource. This can be achieved by adding the value **configure** to the service's **boot.start**_*service* resource. Similarly, adding this value to the service's **boot.stop**_*service* resource will result in the service being stopped as soon as it is removed from the **boot.services** list.

Adding the value **configure** to a service's **boot.reboot**_*service* will result in that service being able to trigger a reboot if it has been started or stopped as a result of a **configure** event.

RESOURCES

Services stop/start

services

List of services (LCFG components and SystemV scripts) to be managed by the component.

levels_*service*

The run levels in which this service should be running.

start_*service*

The start priority level for this service. Also indicates in which boot *event(s)* this service will be started. The default value is **99 restart**.

stop_*service*

The stop priority level for this service. Also indicates in which boot *event(s)* this service will be stopped. The default value is **0 restart**.

reboot_*service*

Indicates for which boot *event* $<s>$ this service will be allowed to trigger a reboot. The default value is **restart**.

Run method

run

List of LCFG components to be called when the boot component's **run** method is invoked.

runmethod_*component*

The method to be called of the given component. The default value is **run**.

user_*component*

The userid to use to call the component. The default value is **root**.

reboot_*component*

Adding `run` to this resource will allow the specified component to trigger a reboot.

Suspend/Resume

suspend

A list of LCFG components to suspend and resume (by calling their **suspend** and **resume** methods). Components are suspended in the order given in the resource, and resumed in the reverse order.

FILES

/var/{lcfg|obj}/tmp/boot.status

This file indicates the current run level and those services which should be running at this level.

/var/{lcfg|obj}/tmp/boot.order

This file indicates the order in which components were started at the last runlevel transitition.

PLATFORMS

Redhat7, Redhat9

AUTHOR

Alastair Scobie <ascobie@inf.ed.ac.uk >

VERSION

1.1.30-1

B.10 client

LCFG profile client

DESCRIPTION

The client profile component for LCFG. This component manages the rdxprof daemon which fetches system configuration protocols from the server (see lcfg-server).

ADDITIONAL METHODS

The **run** method sends a HUP to the daemon initiating a fetch of the profile.

The **context** method is called to change a "context" variable. Arguments of the form *var*=*value* cause the specified context variable to be set to the specified value. A context variable is removed by setting an empty value. The option **-w** before the context arguments can be used to block the method call until the context change is complete and all components have reconfigured.

The **install** method fetches and installs a profile from the URL specified as the first argument (default is the value in the current profile). The optional second argument specifies the filesystem root for the profile installation. Note that this method is designed for use at install time and does not honour the resources which specify **rdxprof** parameters for a client running as a normal daemon. In particular, it does not normally notify components of changes; however an optional **-n** argument can be specified as the first argument to the **install** method, which will be passed to **rdxprof**.

RESOURCES

ack

Set this resource non-null to enable client acknowledgements (**-a** option to rdxprof.

acklimits

The acknowledge time limits **-a** for rdxprof.

components

A space-separated list of components to be notified when their resources change. The default value for this resource references the **profile.components** resource and sorts the entries according to the **ng_cforder** resources of the individual components. It is not normally necessary (and probably a mistake) to manually modify this resource.

debug

A set of rdxprof debug flags.

notify

Non-null for rdxprof to notify components of resource changes by calling the methods defined in the **client.reconfig_***component* resources. (**-n** option).

poll

The poll (**-p**) argument for rdxprof.

rpminc

If this resource is non-null and specifies the pathname of a file which exists (at the time the profile is parsed), then a line will be added to the end of the rpmcfg file to include this file. This is useful for locally specifying additional RPMs.

runupdate *component method component method!client resource*

This resource specifies a method to run when the RPM list changes (rdxprof **-U** option). The method may be followed by any necessary options.

timeout

The HTTP request timeout interval for the rdxprof **-t** option.

url

The list of URL roots for the rdxprof **-u** option. The list may be space-separated (or comma-separated).

verbose

Non-null for rdxprof verbose logging.

warn

rdxprof warning flags.

xmldir

The xml directory for the **-x** option of rdxprof. The default is @XMLDIR@.

PLATFORMS

Redhat7, Redhat9, Solaris9

AUTHOR

Paul Anderson <dcspaul@inf.ed.ac.uk >

VERSION

2.1.35-1

B.11 cron

LCFG cron component

DESCRIPTION

This object configures the `cron` daemon. It does not start the `cron` daemon (this is done out-with the lcfg system), but populates the cron configuration files and signals the running `cron` daemon that they've changed. It will, however, restart the daemon if it finds that it isn't currently running.

Authorization files are constructed for `cron` and `at`. The `cron` object then deletes the existing crontab files for any users who have base crontab files in the directory specified by the `crontabs` resource, or who have an `additions` resource. Base crontabs are then copied in from the crontabs directory, and any additional entries specified by `additions` resources are added.

The `manual` method will call the cron files in the `/etc/cron.*` directories. This is typically used on portables where `crond` is not normally run and this method is called by the manual user update process (via `boot.run`).

allow

A (space-separated) list of users or netgroups for the `cron.allow` file.

deny

A (space-separated) list of users or netgroups for the `cron.deny` file.

atallow

A (space-separated) list of users or netgroups for the `at.allow` file.

atdeny

A (space-separated) list of users or netgroups for the `at.deny` file.

crontabs

A directory containing base crontabs. Any crontabs in this directory will replace the corresponding crontabs on the machine.

additions

A (space-separated) list of *tags* for additional crontab entries specified in the resource database.

add_*tag*

The crontab entry for the specified tag. If the minute field is specified as `AUTO` , the field will be replaced by the machine's host address modulo 60. This is useful for clones.

owner_*tag*

The username under which the crontab entry for the specified tag should be run.

objects

A space-separated list of objects to be run from `cron`. A `cron.run_`*obj* resource must be present for each object listed. The object is executed with the method specifed by the `cron.method_`*obj* resource at the time specified by the `cron.run_`*obj* resource.

run_*obj*

The time at which to run the specified *object* (in `crontab` format). If the minute field is specified as `AUTO` , the field will be replaced by the machine's host address modulo 60. This is useful for clones.

user_*obj*

The username under which to run the specified `object`.

method_*obj*

The method to call for the specified `object`.

args_*obj*

Additional arguments to supply when running the specified *object*.

AUTHORS

```
Jeremy Olsen <J.Olsen@ed.ac.uk>
```

VERSION

1.1.4-1

B.12 dhclient

A component to configure dhclient

DESCRIPTION

This component is really just a wrapper to play with spanning maps, thought it might be used for something else someday. All this components resources are passed to dhcpd servers via spanning map.

RESOURCES

hostname

The hostname of this machine

mac

The mac address of this machine, this must be a valid mac address in the form XX:XX:XX:XX:XX:XX or XX-XX-XX-XX-XX-XX and is validated as the profile is compiled.

hostinstallroot

The installroot to use.

hostfilename

Client bootfile to download, this is usually a bootloader like pxegrub or pxelinux

hostbootmenu

Configuration file for pxegrub.

hostrootpath

Root filesystem to be nfs mountedi, usually at install time.

mailmanager

Boolean resource used to indicate if the manager is to be emailed about problems.

manageremail

The email address that should receive problem reports.

AUTHORS

```
Iain Rae <iainr@dcs.ed.ac.uk
```

VERSION

0.91.15-1

B.13 dialup

LCFG dialup component

DESCRIPTION

This component uses **wvdial** to establish a dialup PPP connection. The configuration file for wvdial is created from the scheme parameters managed by **lcfg-schemes** and the **lcfg-divine** component is used to configure the network parameters once the connection is established.

The **run** method initiates a connection. An optional argument specifies the scheme to use. If no argument is specified, the scheme **def_dialup** is used.

Two special values can be used for resources which refer to wvdial parameters:

<blank>

This value generates a configuration file line for the resource with an empty value. This is different from omitting the line, because wvdial will use the default value for parameter if the line is omitted.

<default>

This is equivalent to leaving the resource value blank; ie. wvdial will use any default value. However, specifying this value prevents the scheme editor nse from closing up blank entries in multi-field values.

RESOURCES

schemes

A (space-separated) list of *scheme tags*. This should include at least one scheme conventionally named `def_dialup` which specifies the default values.

userfile

This resource specifies a (space-separated) list of files containing scheme data that will be read before (and take precedence over) the schemes specified in the resources. This is intended to allow a user to create temporary schemes, eg. while travelling with a portable. These files are normally managed with the scheme editor **nse**. Ability to change this file gives a user the equivalent of root permission. The variable %HOME is substituted with the home directory of the user at the console, so typical values might be: `%HOME/.schemes` or `/home/owner/.schemes`.

modem_<tag>

The full pathname of the modem device. The value <auto> can be used to autodetect the modem using wvdialconfig. (see wvdial man page)

baud_<tag>

Modem baud rate. The value <auto> can be used to autodetect the baud rate using wvdialconfig. (see wvdial man page)

phone_<tag>

Telephone numbers. (see wvdial man page)

init_<tag>

Modem initialisation strings. The value <auto> can be used to autodetect the init strings using wvdialconfig. (see wvdial man page)

username_<tag>

Dialup username. (see wvdial man page)

password_ <**tag** >

Dialup password. (see wvdial man page)

dial_ <**tag** >

Modem dial command. (see wvdial man page)

wvdial_ <**tag** >

Additional wvdial parameters. (see wvdial man page)

PLATFORMS

Redhat9

AUTHOR

Paul Anderson <dcspaul@inf.ed.ac.uk >

VERSION

0.99.12-1

B.14 divine

Network configuration component for LCFG.

DESCRIPTION

This object configures and controls the divine network probe. The `probe` method is called when one of the controlled interface comes, up to probe the network and determine the appropriate ”scheme”. The interface is set accordingly, and affected LCFG components (such as mail) are reconfigured by using the context mechanism. The `run` method can be used to initiate a probe manually.

If an interface is a wireless interface, then all the specified wireless network names will be probed, in the order that they appear in the scheme list. All schemes corresponding to a particular network will be probed in parallel. If the interface is not a wireless network, then all non-wireless schemes will be probed in parallel. Default schemes will be tried if all probes fail. DHCP is used if no explicit IP address is given.

Most addresses can be specified as DNS names or numbers. However, if the hostnames are not in the local DNS, they must be specified as numbers, otherwise they will not be available when booting on remote networks.

OPTIONS

The following options are supported by the `start`, `run` and `probe` methods:

-a *count*

The number of times to retry an arp request when probing the network. Small values may fail to detect networks with a slow arp response. Large values will increase the time required to probe a series of networks. The default value is set by the **arptries** resource.

-C

Output progress to /dev/console.

-D

Debugging.

-e *count*

The number of times to retry a DHCP request. Small values may fail to detect networks with a slow DHCP response. Large values will increase the time required to probe a series of networks. The default value is set by the **dhcptries** resource.

-R *reason*

The given reason for the network probe is noted in any informational messages. This set to the interface name, for example, when a probe occurs because the interface is coming up.

-s *scheme*

Attempt to set the specified scheme (only). No attempt is made to probe or to determine the applicability of the scheme. The scheme is attempted on all interfaces in the **if** resource.

-t *seconds*

The timeout on DHCP requests. Small values may fail to detect networks with a slow DHCP response. Large values will increase the time required to probe a series of networks. The default value is set by the **dtimeout** resource.

-v

Verbose. Displays all probe attempts.

-w *seconds*

The timeout on wireless access point detection. Small values may fail to detect networks with a slow AP response. Large values will increase the time required to probe a series of networks. The default value is set by the **wtimeout** resource.

RESOURCES

arptries

The number of times to retry an arp request when probing the network. Small values may fail to detect networks with a slow arp response. Large values will increase the time required to probe a series of networks. The default value is 6.

dhclient

Use dhclient instead of pump for DHCP. (NOT YET IMPLEMENTED).

dhcptries

The number of times to retry a DHCP request. Small values may fail to detect networks with a slow DHCP response. Large values will increase the time required to probe a series of networks. The default value is 2.

dtimeout

The timeout on DHCP requests. Small values may fail to detect networks with a slow DHCP response. Large values will increase the time required to probe a series of networks. The default value is 2.

interfaces

A (space-separated) list of interfaces to be controlled by divine. Each interface will be tried in the given order and the first interface which matches a particular scheme will be set accordingly. All other interfaces will be disabled.

type_*interfaceinterface!divine resource*

The type of the specified interface. If the interface is supported by the ”linkstatus” command, then this can be used to determine whether or not the interface cable is connected. This means that the schemes can be probed faster because there is no need to wait for ARP timeouts on disconnected interfaces. Supported types are ”mii” and ”eepro100”. The default is null (interface type unknown), in which case no test is made for the presence of the cable.

schemes

A (space-separated) list of *scheme tags*. This should include at least one scheme conventionally named `default` which specifies the normal default network parameters.

pidfiles

A (space-separated) list of filenames assumed to contain process IDs. When the scheme changes, each process will be sent a USR2 signal and the name of the new scheme will be in `/var/lcfg/tmp/schemes.scheme`. This can be used by processes to monitor and display scheme changes (eg. the ”sleepbutton” provided with obj-kdm). The variable %HOME is substituted with the home directory of the user at the console, and the process kill is run under the uid of the pidfile owner. The default is %HOME/.schemes.pid.

route

If this resource is true, divine will add a default for route for the specific gateway. If it is false, it will generate resources for a routing component to handle the routing.

userfile

This resource specifies a (space-separated) list of files containing scheme data that will be read before (and take precedence over) the schemes specified in the resources. This is intended to allow a user to create temporary schemes, eg. while travelling with a portable. These files are normally managed with the scheme editor **nse**. Ability to change this file gives a user the equivalent of root permission. The variable %HOME is substituted with the home directory of the user at the console, so typical values might be: `%HOME/.schemes` or `/home/owner/.schemes`.

defcontext

If this resource is non-null, it should be a profile ”context” which will be used for schemes which don’t specify an explicit context. (This is in addition to the **scheme=***name* context).

nocontext

If this resource is non-null, it should be a profile ”context” which will be set when no network scheme can be detected. (This is in addition to the **scheme=** context).

dohosts

If this resource is true, then the IP address is registered in /etc/hosts as the address of the host.

oksound

The name of a sound file to play when a network scheme is successfully selected.

failsound

The name of a sound file to play when no network scheme can be selected.

cfopts

The options to be applied to divconf when it is called because an interface has come up.

wtimeout

The timeout on wireless access point detection. Small values may fail to detect networks with a slow AP response. Large values will increase the time required to probe a series of networks. The default value is 0.5.

descr_<tag>

An optional description of the scheme.

if_<tag>

A (space-separated) list of interfaces for which this scheme is appropriate. If this field is blank, the scheme is a candidate for all interfaces. Use ”ppp0” for schemes which are intended only for dialup.

wnet_<tag>

The name of a wireless network to which the scheme should be applied. If this is blank, the scheme applies only to wired interfaces. The name may be prefixed with /ad-hoc for an ad-hoc network (default is ”/managed”).

aplist_<tag>

A (space-separated) list of access point MAC addresses for which this scheme is valid. By default, schemes are valid for any AP. Mac Addresses must have the form XX:XX:XX:XX:XX:XX.

wep_<tag>

The WEP encryption key for this wireless network as a hex string, or in the form ”s:PASSWORD”. If this is blank, no encryption is performed. This field is ignored for wired networks.

masksize_<tag>

The size (in bits) of the netmask for this network (default 24). If DCHP is being used (IP field is blank) or the scheme is being used for PPP, then this is determined automatically.

script_<tag>

The name of script to execute when this scheme is detected. The name of the scheme is passed as an argument, and the script is run with the uid of the user owning the scheme file. The script is executed in the background with input and output to /dev/null.

smtp_<tag>

The name of a host to use as the mail relay. Local mail will always be delivered via the local sendmail program which will forward it to this relay. If this is blank, then the LCFG-defined default mail relay will be used. Setting this to ”localhost” will cause the local sendmail to deliver mail directly; this is usually necessary when connected to a foreign network.

context_ <tag >

A context to be passed to the LCFG profile component. This can used to tune the system configuration towards the type of connection. The default is set by the defcontext resource (normally "net=remote"). The additional context "scheme=ID" is also selected.

ip_ <tag >

The IP address (or hostname) to which the host should be configured if this scheme is detected. If this is blank, then DHCP will be used in an attempt to determine the host address. If a name is specified in the LCFG (rather than an IP number), then the name must be available in the local DNS server. This field is ignored for dialup schemes and the IP address is determined via PPP.

probe_ <tag >

The IP address (or hostname) of one or more well-known hosts (eg. the gateway) to be probed to identify this scheme. If this is blank, the scheme will be applied as a default if all others fail. Note that more than one default scheme is not meaningful, unless they are on different wireless networks. If a name is specified in the LCFG (rather than an IP number), then the name must be available in the local DNS server. This field is not used for dialup connections.

gw_ <tag >

The IP address (or hostname) of the gateway corresponding to this scheme. This value is set automatically if DHCP is being used (IP field is blank), or when using a dialup scheme. If a name is specified in the LCFG (rather than an IP number), then the name must be available in the local DNS server. The special value <route > indicates that the existing default values for the routing component will be used to set up the routing.

dns_ <tag >

The IP addresses (or hostnames) of DNS servers to use with this scheme. These hosts will be used as forwarders by the local DNS server which is always queried first. If this is blank, then the default forwarders will be used. If it is *, then no forwarders will be set. If a name is specified in the LCFG (rather than an IP number), then the name must be available in the local DNS server. If left blank, DHCP and PPP will supply these values automatically.

PLATFORMS

Redhat7, Redhat9

AUTHOR

Paul Anderson <dcspaul@inf.ed.ac.uk >

VERSION

3.5.31-1

B.15 dns

The LCFG DNS component

DESCRIPTION

This component starts the DNS service. It generates the DNS client configuration (`/etc/resolv.conf`). If the resource `dns.type` is set to `server` it also generates the server configuration (`/etc/named.conf`) and starts the server. The `update` method schedules immediate zone maintenance for some or all of a server's configured zones.

GENERIC RESOURCES

type

The type of DNS service. Valid options are `client` (the default) and `server`.

contextlabel

This resource does not actually affect the operation of the component, but instead is included in some of its messages. Setting it to some lcfg context-specific value might therefore be useful to the user.

logFile

This resource defines the name of a log file, which will be processed when the logrotate method is called.

RESOLVER RESOURCES

ourdomain

What domain do we live in? (We can't rely on hostname or domainname or dnsdomainname or the like for this, as they're likely to try to do some kind of address lookup and we can't rely on that working!)

servers

A list of servers to place in the `/etc/resolv.conf` file. The order of servers in the list can be randomized. If `type` is set to `server` then `servers` will default to 127.0.0.1. Note that while the object will translate names to the addresses required in the configuration file, this will be done using the `/etc/resolv.conf` file's **previous** contents. It might therefore be thought better for this resource to contain explicit addresses rather than names.

randomize

This resource, if set to `yes`, will randomize the `dns.servers` list.

fallback

A list of servers to be used in extremis if `servers` happens not to be set for some reason. Dotted-quads would probably be a good idea here. The order of these won't be randomised.

options

A list of `resolv.conf` options.

search

A list of domains for the `resolv.conf` "search" list.

global_sortlist

cluster_sortlist

local_sortlist

Sortlists to be included in the `/etc/resolv.conf` file. "local" entries come first, followed by the machine's attached wires, with the "global" entries coming last.

local_netmask

A netmask to be applied to the machine's attached interfaces when constructing the sortlist.

explicit_sortlist

If set, only the explicit sortlist resources are used when constructing the resover sortlist. The implicit list derived by the component from the configured interfaces is **not** used.

SERVER RESOURCES

forwarders

The addresses of forwarders which should be queried for unknown names before going out onto the Internet at large.

slave

If forwarders are set, use them exclusively to answer for unknown names and don't ever ask on the Internet at large.

transfers_in

If set, limits the number of concurrent inbound zone transfers. If not set the compiled-in version-dependent default is used.

files

If defined, set an upper bound on the number of files which the server is allowed to have open at any one time. Usually this is set high as a back-stop.

notify

Tell all the NS-listed nameservers when a zone is changed. They'll still eventually find out anyway through the usual zone-maintenance mechanisms, but this speeds things on a little. Note that it is also possible to specify this on a per-zone basis.

also_notify

Contains a list of addresses of stealt-secondary nameservers which should be notified when a master zone changes.

query_source

What should the source address of queries made by the nameserver look like? (Normally this is used to fix the source port for firewalling; the default is to use an unspecified anonymous one.)

transfer_source

Specify the source address and/or port to be used for zone transfer requests. If not specified the default is to use any arbitrary port $>$1024.

run_user

run_group

Specify the user and/or group which the server should run as so as to limit any security exposure which might arise. The component will attempt to chown any files and directories as necessary.

umask

The umask which the component should use, and which will be inherited by any processes it starts.

pid_file

The name of a file into which the nameserver's pid is written at startup.

version

How should the server answer "`version.bind txt chaos`" queries? If this is blank then the compiled-in default (usually the software version) is used. If it's "RCS" then the dns component's RCS ID is used. Anything else is used verbatim.

listen_on

If set, contains a list of interface addresses on which `named` will listen for requests. (127.0.0.1 is the most likely value for this resource to be set to.)

dialup

If set, causes normal zone maintenance to happen only at heartbeat intervals. This can avoid bringing up dialup lines or making large zone transfers over slow links.

heartbeat_interval

How often to do "dialup" zone maintance. The compiled-in default is 60 (minutes). Setting this to zero disables automatic zone maintenance, so updates are only done after an explicit request.

interface_interval

How often should `named` scan for new or departing interfaces? The compiled-in default is usually reasonble.

channels

categories

Define the logging done by the nameserver.

`channels` contains a list of channel tags. For each tag there's a corresponding `channel_whatever` resource that contains the body of the clause to be written to the configuration. Likewise, `categories` contains a list of tags for `category_whatever`.

zones

`zones` contains a list of zone tags for the zones carried on this server For each tag in `zones` there are corresponding `type_...`, `file_...` and `masters_...` resources. The component applies "reasonable" rules as to whether these are required or not. Each zone also has require `zone_...` and optional `znotify_...` resources.

updates

`updates` contains a list of all the defined update-sets. For each entry there's a corresponding `update_thing` which contains a list of zone tags. The first entry in `updates` is used by default if no user-supplied parameter is passed to the Update() method.

acls

`acls` contains a list of tags specifying which access control list entries to configure in to the `/etc/named.conf` file. For each tag there is a corresponding `acl_...` resource containing a list of values, in one of bind's acceptable formats, defining the contents of the acl entry. The tag value is used as the name of the acl itself.

allow_query

Contains a list of networks or acl-names, in standard `bind` format, which are allowed to query this nameserver. An empty list means no restriction.

allow_transfer

Contains a list of networks or acl-names, in standard `bind` format, which are allowed to do zone-transfers from this nameserver. An empty list means no restriction.

allow_recursion

Contains a list of networks or acl-names, in standard `bind` format, which are allowed to make recursive queries through this nameserver. An empty list means no restriction.

allow_notify

Contains a list of networks or acl-names, in standard `bind` format, which are allowed to send notify messages to this nameserver. An empty list means no restriction.

named

Where to look for the `named` binary itself.

rndc

Where to look for the `rndc` control program.

pending

A list of IN-class files in `named.conf` format, to be included in the generated server configuration file. The `pending` method will rotate any new versions of the files on this list into place. How those new versions get there is outwith the scope of this component, though an example `expect` script is distributed with it.

serial_query_rate

Used to limit the number of outstanding SOA queries during zone maintenance. The value is in queries/second.

zoneStats

Set to enable per-zone statistics.

statistics_file

Specifies the name of the file into which the server will dump its statistics on request.

dump_file

Specifies the name of the file into which the server will dump its internal database on request.

lwres

Enable lightweight resolver support in the server.

INview_match

The "match" rules which should apply to the IN-class views which the component generates in the `/etc/named.conf` file.

INSTALLATION RESOURCES

The following resources are used only by the component's Install() method, and therefore do not have any effect in during normal operation.

installservers

A list of servers to use in addition to any passed in as parameters to the Install() method.

installsortlist

The sortlist, if any, to be defined in the install-time `/etc/resolv.conf` file.

installinterface

The name of the interface whose address and netmask should be used to compute the sortlist for the install-time `/etc/resolv.conf` file if one is not specified explicitly.

PRIVATE RESOURCES

The following resources should not normally have their values changed from the installation defaults. They define where the component's various helper programs have been installed, or to provide Solaris/Linux compatibility hooks. Setting them incorrectly may result in the component not functioning correctly. Refer to the component source itself for details as to their various functions.

keygen

srvgen

makesortlist

getaddr

shufflestring

PLATFORMS

RedHat 7, RedHat 9. Previous versions ran on Solaris 2.6.

AUTHORS

```
George Ross <gdmr@dcs.ed.ac.uk>
```

VERSION

6.1.39-1

B.16 example

An example LCFG component

DESCRIPTION

This component is an example only.

RESOURCES

server

An example resource which gets substituted into the configuration file.

PLATFORMS

Redhat7, Redhat9, Solaris9

AUTHOR

Paul Anderson <dcspaul@inf.ed.ac.uk >

VERSION

1.1.4-1

B.17 file

The LCFG file component

DESCRIPTION

The file component can be used to create arbitrary configuration files and directories without the need to write a custom component. The templates for the configuration files, and the resources to be substituted in them, may be supplied directly as resources of the `file` component, or may be specified in an independent `.def` file (a "managed component").

RESOURCES

components

A list of "managed" components for which configuration files should be generated. For each component *comp* in the list, the resources specified below (see MANAGED COMPONENT RESOURCES) must be present. This list should normally include the `file` component itself, so that configuration files can be generated from `file` resources without any additional default files. The default value is `file`.

files

See MANAGED COMPONENT RESOURCES

file_*tagtag!file resource*

See MANAGED COMPONENT RESOURCES

group_*tagtag!file resource*

See MANAGED COMPONENT RESOURCES

mode_*tagtag!file resource*

See MANAGED COMPONENT RESOURCES

owner_*tagtag!file resource*

See MANAGED COMPONENT RESOURCES

tmpl_*tagtag!file resource*

See MANAGED COMPONENT RESOURCES

type_*tagtag!file resource*

See MANAGED COMPONENT RESOURCES

variables

A list of variable tags for variables to be substituted in any templates declared in the `files` resource.

v_*varvar!file resource*

The values of the variable given by the `var` tag.

MANAGED COMPONENT RESOURCES

Each managed component should define the following resources to specify the configuration files or directories to be created on its behalf by the `file` component. The managed components may include code of their own but, typically, they do not - they consist of only a `.def` file and the `file` component uses this to create their configuration files.

Managed components should also set the resource *comp***.ng_cfdepend** to >**file** so that the file component is called whenever the resources change.

comp.**files**

A list of file/directory tags. Each tag corresponds to a configuration file or directory to be created on behalf of the component *comp*.

comp.**file**_*tag*

A space-separated list of pathnames for configuration files or directories corresponding to *tag* for component *comp*. Normally, these will be full pathnames, but several objects in the same directory can be specified by giving a common prefix; for example:

```
/home/users : fred jill
```

comp.**group**_*tag*

The group for the file or directory corresponding to *tag*. If this is null, then files are created with the same group as the template, and directories with the same group as the running component.

comp.**mode**_*tag*

The file mode for the file or directory corresponding to *tag*. If this is null, then files are created with the same mode as the template, and directories with the mode 0755.

comp.**owner**_*tag*

The owner for the file or directory corresponding to *tag*. If this is null, then files are created with the same owner as the template, and directories with the same owner as the running component. Changing ownership of links normally changes the ownership of the target and is not recommended. If the owner is specified as `*`, then the name of the file itself is used as the owner. This is useful for creating home directories, for example.

comp.**tmpl**_*type*

If the file corresponding to *tag* has type `template`, then this resource should give the full pathname of the template file. If the type is `literal`, then this resource should specify the template literally - newlines may be included by using the `\n` escape sequence.

The template is passed through the template processor with the resources of *comp* defined, to create the configuration file.

comp.**type**_*type*

The type of the configuration file or directory corresponding to *tag* for component *comp*. This may be `delete`, to delete any existing configuration file, `template` to create a file from a template file, `literal` to create a file from a literal template, `link` to create a symlink from the template to the file, or `dir` to create a directory. The type may optionally be followed by a colon and a space-separated list of options. The only option currently supported is `zap` which will delete any existing filesystem object with the same pathname as the target file if it has a different type. Directories will be deleted recursively and this should be used with care!

MANAGED COMPONENT EXAMPLE

```
#include "mutate.h"
#include "ngeneric-1.def"

!ng_cfdepend      mSET(>file)
!ng_statusdisplay mSET(false)
!ng_reconfig      mSET()

@files type_$ tmpl_$ file_$
type_$
tmpl_$
file_$
owner_$
group_$
mode_$
var1
var2
```

```
files A B homes

type_A template
tmpl_A path to the template
file_A path to the config file
type_B literal
tmpl_B the value of var1 is <%var1%>
file_B path to the config file

var1 some value to substitute in a template
var2 some value to substitute in a template

type_homes dir
file_homes /home : fred jill
owner_homes *
group_homes users
mode_homes 0700
```

PLATFORMS

Redhat7, Redhat9, Solaris9

AUTHOR

Paul Anderson <dcspaul@inf.ed.ac.uk >

VERSION

1.0.9-1

B.18 foomatic

The LCFG foomatic component

DESCRIPTION

This component configures printers using **foomatic**. It also (optionally) configures **printcap**, **lpd.conf**, and **lpd.perms**. This component does not manage printer daemons - use another component, such as **lprng** in conjunction with **foomatic**.

RESOURCES

attr_*entryentry!foomatic resource*

The attribute name for the printcap entry corresponding to tag *entry*.

conf

A list of tags for variables to be substituted in the **lpd.conf** template.

conf_*varvar!foomatic resource*

The value of a configuration variable to be substituted in the **lpd.conf** template.

conftmpl

The pathname of a template used to create the **lpd.conf** file. If this is null, any existing **lpd.conf** file is unchanged.

connect_*qq!foomatic resource*

The foomatic connection parameter for the queue specified by the tag *q*. Foomatic is alleged to support the following:

```
file:/path/file                 # includes usb, lp, named pipes
ptal:/provider:bus:name         # HPOJ MLC protocol
lpd://host/queue                # LPD protocol
socket://host:port              # TCP aka appsocket
ncp://user:pass@host/queue      # Netware (LPD, LPRng, direct)
smb://user:pass@wgrp/host/queue # Windows
stdout                          # Standard output (direct)
postpipe:"<command line>"       # Free-formed backend command line
```

If this does not work for you, you can use **file:/dev/null** and then use the **pcap** resource to replace the **lp** parameter in the printcap with whatever you like. NOTE: if the connect resource is not set, **foomatic-configure** will not be called for this queue. This allows special queues to be hand-crafted into the printcap using the **pcap** resources.

default

The tag corresponding to the print queue to be used as the default.

descr_*qq!foomatic resource*

The foomatic description parameter for the printer queue corresponding to the tag *q*. (see man foomatic-configure).

driver_*qq!foomatic resource*

The foomatic driver parameter for the printer queue corresponding to the tag *q*. (see man foomatic-configure).

location_*qq!foomatic resource*

The foomatic location parameter for the printer queue corresponding to the tag *q*. (see man foomatic-configure). Note that this must be unique among all the print queues!

opt_*oo!foomatic resource*

The foomatic name of the specified option.

options_*qq!foomatic resource*

A list of tags for any options to be supplied to foomatic for this printer queue. Read **/etc/foomatic/lpd/*q*.lom** to see the available options and values.

optv_*oo!foomatic resource*

The value for the specified option.

pcap_*qq!foomatic resource*

A list of tags for additional printcap entries that should replace or augment those generated by foomatic for the specified printer.

pcaptmpl

The name of the printcap file generated by foomatic.

perms

A list of tags for variables to be substituted in the **lpd.perms** file.

perms_*varvar!foomatic resource*

The value of a variable to be substituted in the **lpd.perms** file.

permstmpl

The pathname of a template used to create the **lpd.perms** file. If this is null, any existing **lpd.conf** file is unchanged.

printcap

The name of the printcap file used by the print system. ie. created by the foomatic component from the file generated by foomatic.

printer_*q*_*q!foomatic resource*

The foomatic printer parameter for the printer queue corresponding to the tag *q*. (see man foomatic-configure).

queues

A list of printer queues to be configured.

spooler_*qq!foomatic resource*

The foomatic printer parameter for the printer queue corresponding to the tag *q*. (see man foomatic-configure).

value_*entryentry!foomatic resource*

The value of the printcap entry corresponding to tag *entry*. The special values <**true** > and <**false** > can be used to define a binary parameter with no value, and to delete a parameter created by foomatic, respectively.

PLATFORMS

Redhat9

AUTHOR

Paul Anderson <dcspaul@inf.ed.ac.uk >

VERSION

0.99.9-1

B.19 fstab

LCFG fstab component

DESCRIPTION

The object partitions disks and maintains the `/etc/fstab` file.

The `/etc/fstab` is first constructed from the partition information specified for the hard disks described by the `disks` and `partitions`_*disk* resources. The `entries` resource is then used to add additional `fstab` entries.

The `adddisk` method is used to add a new disk to the system. It will (optionally) partition the disk, make the filesystems and create `/etc/fstab` entries. It is called implicitly by the `preparedisks` method at system install time, but can be called manually thereafter to add new disks.

The `mountdisks` and `umountdisks` methods can be used to mount or unmount all configured disks. These are only intended for use when running the installroot for machine debugging/patching.

disks

A list of disks attached to this machine.

dopartition_*disk*

This resource, if set to `no`, will stop the component from partitioning the disk *disk*.

partitions_*disk*

A list of partitions for *disk*. A disk can only have primary partitions (up to 4). Extended and logical partitions are not yet supported - the disk should be partitioned manually if these are required.

size_*partition*

The size of the specified partition. The partition size can be given in megabytes, or be set to `free` to indicate that the partition should use up remaining disk space or be set to <existing > to indicate that the partition size and location should not be changed.

type_*partition*

The type of filesystem of the specified partition. Currently supported partition types are `ext2`, `ext3`, `raid` and `swap`. Alternatively, a numeric ID can be used to specify the partition type.

mpt_*partition*

The mount point for the filesystem on the specified partition.

mkmpt_*partition*

Determines how the component should behave when the mount point already exists and is populated with files. Setting this resource to `fail` will cause a failure, `ignore` will ignore prexisting files and `zap` will delete any prexisting files.

mntopts_*partition*

Describes any mount options for the filesystem associated with the specified entry or partition.

passno_*partition*

Describes the fsck passno for the specified entry or partition.

preserve_*partition*

This resource, if set to `yes`, indicates that the component should not rebuild the filesystem when the `adddisk` method is invoked.

mkprog_*partition*

The program to create the file system (or whatever) in the specified partition.

mkopts_*partition*

Any options to pass to the appropriate tool (eg mke2fs) for building the specified partition.

entries

A list of additional <fstab > entries.

spec_*entry*

Describes the block special device or remote filesystem associated with the specified entry.

file_*entry*

Describes the mount point for the specified entry.

vfstype_*entry*

Describes the type of filesystem for the specified entry.

mntopts_*entry*

Describes any mount options for the filesystem associated with the specified entry or partition.

freq_*entry*

Describes the dump frequency for the specified entry.

passno_*entry*

Describes the fsck passno for the specified entry or partition.

updfstab

A list of extra entries for `/etc/updfstab.conf`. An entry can be either a device, `upfdef_`*`tag`* or an include file of further definitions `updfile_`*`tag`*.

updfdev_*tag*

A list of line references for the specified device.

updfline_*tag*_*line*

An individual line for the specified device.

updfile_*tag*

AUTHORS

`Alastair Scobie <ascobie@inf.ed.ac.uk>`

VERSION

1.1.22-1

B.20 gdm

LCFG gdm component

DESCRIPTION

This component configures the Gnome display manager.

RESOURCES

allowremoteroot

True to allow remote root logins.

allowroot

True to allow local root logins.

autologin

The name of a user to be logged in automatically on startup.

bgcolor

The background colour for the login screen (if **bgtype** is set to 2).

bgimage

The background image for the login screen (if **bgtype** is set to 1).

bgscale

True to scale the backgound image to fit the screen.

bgtype

```
0 = no background
1 = image background
2 = solid colour background
```

broadcast

True to broadcast for XDMCP.

browser

True to display user browser.

command_*tagtag!gdm resource*

The shell command to run for the named *tag*.

commands

A list of tags for commands to run during the display of the login screen.

configavailable

True to allow the configuration to be changed from the login screen. Useful only for testing, since this will be overwritten when the component reconfigures.

defaultface

The icon for the default face in the browser.

defsession

The tag for the default session if the user has not specified a preference.

exclude

A comma-separated list of usernames to be excluded from the browser.

facedir

The directory containing face icons for the browser.

greeter

The name of the greeter program. The graphical greeter is not (yet) supported and you probably don't want to change this.

haltcommand

The command to halt the system.

honorindirect

True to honor indirect XDMCP queries.

hosts

A comma-separated list of hosts to add to the chooser.

initcmd

A shell command to execute (as root) when the display manager is initialized.

logo

The full pathname of an image file to use as the logo in the greeter.

menuname_*tagtag!gdm resource*

The name to appear in the menu for the session corresponding to *tag*.

minuid

The minimum UID for users to appear in the browser.

precmd

A shell command to execute (as root) before the user session runs.

postcmd

A shell command to execute (as root) after the user session has ended.

rebootcommand

A shell command to reboot the system.

servers

A list of display numbers to run servers on.

session_*tagtag!gdm resource*

The shell command to execute for the session corresponding to *tag*.

sessioncmd

A shell command to execute (as the user) when the user session starts.

sessions

The list of tags for the sessions to be displayed in the menu.

suspendcommand

A shell command to suspend the system.

systemmenu

True to display the system menu on the login screen.

titlebar

True to display a titlebar on the login screen.

welcome

The text string for the welcome message. The string %n is replaced with the hostname, and \n can be used for a multi-line message.

x

The x-coordinate of the greeter box. Negative values can be used to provide an offset from the right of the screen. If both this and **y** are empty, then the greeter will be centered.

xdmcp

Enable XDMCP. Do not do this witout setting appropriate entries in hosts.deny or hosts.allow. See the GDM manual.

y

The y-coordinate of the greeter box. Negative values can be used to provide an offset from the bottom of the screen. If both this and **x** are empty, then the greeter will be centered.

AUTHOR

Paul Anderson <dcspaul@inf.ed.ac.uk >

PLATFORMS

Redhat9

VERSION

0.99.20-1

B.21 grub

Component to generate and install grub bootloader

DESCRIPTION

This component is used to generate grub menu (.lst) files both to configure grub on the local machine and to provide .lst files for remote hosts to boot off via pxegrub. The object supports both the VGA and serial consoles and will allow arbitrary command line options to be passed to the kernel.

The component builds up a series of files based on the **grubfiles** resource and stored them in /tftpboot/grub_files. A local configuration file can be specified for install into /boot/grub/menu.lst.

RESOURCES

localconf

The **grubfile** configuration which should be used as the local configuration file, this will be copied to /boot/grub/menu.lst

grubfiles

A list of configuration files which the grub component should generate, each file is made up of a number of global configuration statements followed by a list of operating systems to boot.

defaultboot_$

The default menuitem to boot (0 is the first.....).

timeout_$

how long to wait until booting the default menuitem

fallback_$

The menu item to fall back to if the default menu will not boot.

menucolour_$

A pair of colours (foreground/background) for the menu.

menucolourselect_$

Similarly a pair of colours to set a selected menu item to.

serialunit_$

This resource is used to define the serial port to be used when grub is to be used via a serial console, 0=com1, 1=com2 .....

serialspeed_$

The speed to set the coms port to.

terminal_$

Where to display output from grub, this can be console (on a PC the SVGA port) or serial (the serial port selected by **serialunit** or both serial and console in which case the first connection which returns keypress will be selected.

menulist_$

A list of menuitems to be used in this grub file.

hiddenmenu_$

This resource is used to replace the standard grub menu with something a bit more cryptic.

password_$

A password to protect the menu. This should be an md5crypt generated by running /sbin/grub and using the md5crypt command. This prevents users from editing the menu items by using the e option.

menuitems

A tag list of menu items, these are essentially different boot options.

tite_$

Some identifiable title for the item.

lock_$

Prevents anyone booting a menuitem without entering a password

mpassword_$

Password to control access to this menu item, this can either prevent people editing the item or (in conjunction with the **lock** resource, prevent anyone booting an item. As with the **password_$** resource this is an md5cryp.

root_$

The grub root device, not to be confused with a Unix root filesystem.

configfile_$

Specify an alternative configuration file to use.

chainloader_$

Grub normally can deal with booting OS's but in cases where it can't this simply treats the appropriate chunk of disk like a boot record and attempts to boot it. =item **kernel_$**

The path to the kernel to use (usually /boot/vmlinuz).

kroot_$

the root filesystem to boot from.

kernelargs_$

Any kernel arguments to be passed to the kernel.

initrd_$

An initial ramdisk to be loaded.

boot_$

Umm, it's all hopefully loaded, boot it.

splashimage

Define a background image to be displayed by grub. NB this is a redhat addition and may not work with official releases of grub.

clientmode

A yes/no flag to indicate whether or not grub should generate a local configuration file and install itself on the MBR of the local machine.

servermode

A yes/no flag to indicate whether the host should generate grub configuration files for use in pxegrub installs.

For more information see the grub manual

AUTHORS

Iain Rae <iainr@inf.ed.ac.uk>

VERSION

1.2.3-1

B.22 hardware

LCFG hardware component

DESCRIPTION

The object configures hardware. It is also expected, eventually, to twiddle hardware parameters for things like disks. It can (primitively) configure PNP devices such as sound cards, etc.

chmoddevices

> A list of chmod commands to set protection on device files.

chmoddevices_*entry*

> The chmod command for tag *entry*.

pnpdevices

> A list of PNP boards to be configured at boot time. The configuration files for each PNP board live in `/etc/obj/conf/pnp`.

modlist

> A list of kernel module rules to be added to the `/etc/modules.conf` file.

mod_*tag*

> The kernel module rule associated with *tag*.

permmodules

> A list of modules to be installed at boot time. Options for the module loader can be specified by use of a modopt_*module* resource. The module name for this module can be overriden by use of a modname_*module* resource - this is useful when the module file lives in a non standard location. The modloader_*module* resource specifies which loader to use.

modopt_*module*

> Module loader options for module *module*

modname_*module*

> This resource can be used to override the name of the module given to the module loader.

modloader_*module*

> Specifies which module loader to use for this module. Defaults to `/sbin/insmod`.

devices

> A list of tags specifying device aliases to be created in /dev.

devalias_*tag*

> The alias to be created for the specified device tag. If this resource is missing, it defaults to the same as the tag.

dev_*tag*

> The name of the device file for the specified device tag.

apm_script

> The name of a script to be called when the `resume` and `suspend` methods are invoked. The particular method used is passed as an argument to this script. This is useful for esoteric laptops that aren't satisfied by the following simple hacks.

apm_vt

If this resource is set, the `suspend` method will change to VT1 (first virtual terminal) prior to the machine being suspended. The `resume` method will change to the virtual terminal specified by this resource. This resource has no effect if the `apm_script` resource is set.

apm_netrestart

If set to "yes", the network is restarted by the `resume` method. This resource has no effect if the `apm_script` resource is set.

tpreset

If set to "yes", a Synaptics Trackpoint or Touchpad will be reset.

videobusmaster

If set to "yes", the PCI or AGP card with the video card will be configured to be a bus master (using setpci)

AUTHORS

`Alastair Scobie <ascobie@inf.ed.ac.uk>`

VERSION

0.100.4-1

B.23 init

LCFG init component

DESCRIPTION

This object maintains local additions to the `/etc/inittab` file. It also applies local hacks to the initscripts.

entries

A list of `inittab` entries to be added to the `/etc/inittab` file.

entry_*tag*

The `inittab` entry associated with *tag*.

AUTHORS

```
Alastair Scobie <ascobie@inf.ed.ac.uk>
```

VERSION

0.100.2-1

B.24 install

LCFG install component

DESCRIPTION

This component controls which components are called at install time.

RESOURCES

installmethods

A list of install method tags to call in sequence to install a machine.

imethod_*tag*

The install method associated with *tag*. Each method can be either an LCFG component call (to be passed to **om**) or one of the following built-in operators.

In the following, *targetroot* is the path to the root of the target system.

%configclock% *targetroot*

Configures the /etc/sysconfig/clock file of the target system.

%gettime% rdate|ntpdate *timeservers*

Sets the system time using either rdate or ntpdate. *timeservers* is a list of servers to query.

%setclock%

Sets the hardware clock from the current system time.

%umount% *targetroot*

Attempts to unmount all mounted filesystems under *targetroot*.

%settz% *targetroot*

Creates the target system's /etc/localtime link. See the **timezone** resource.

%oneshot% *param*

Will **eval** the shell string *param*. The use of this operator is seriously frowned upon; it should only be used for development purposes.

utc

If this value is set to **yes**, indicates that the hardware clock is kept in UTC. Set to **no** if not. Is used both for the **%configclock%** built-in for setting the target system's /etc/sysconfig/clock file and by the **%setclock%** built-in for setting the current time.

timezone

This configures the timezone for the target system by linking the specified /usr/share/zoneinfo file to /etc/localtime.

PLATFORMS

Redhat9

AUTHOR

Alastair Scobie <ascobie@inf.ed.ac.uk >

VERSION

0.100.15-1

B.25 inv

LCFG inventory component

DESCRIPTION

These **inv** resources define the inventory information for an LCFG node. There is no **inv** component; the resources are published to a spanning map which can be subscribed by other components such as **lcfg-inventory**. The **inv** resources are also used by the server for publication on the HTML status pages.

RESOURCES

allocated

A space-separated list of users to which the machine is allocated.

cluster

The name of a spanning map to which the full inventory information should be published. The default is **inventory/all**.

comment

A comment.

date

Date machine initially purchased or installed. Must be of the form dd/mm/yy with yy in the range 80-99 or 00-20.

display

A (space-separated) list of **inv** resource names to be displayed on the server status page. Each name may be prefixed with *label*= to set the label used to display the field (the resource name is used by default). Tilde characters in the label are replaced with spaces to allow labels containing spaces to be specified.

domain

The domain name. Referenced from the profile by default.

location

The location of the machine.

maintainer

Information on the maintenance contract.

manager

A valid username who is responsible for management of the machine.

model

The model of the machine, eg. “Sun IPX” or “Dell Optiplex Gxa”. The first word of the model should be the “make”.

node

The node name. Referenced from the profile by default.

os

A space-separated list list of operating systems running on the machine. The first should be the primary operating system.

owner

The group owning" the machine. Eg. "lfcs" or "cs".

shortlist

The name of a spanning map to which short information should be published. This is not used by default.

sno

The serial number.

tags

A (space-separated) list of keywords for identifying different properties or groups of hosts. The keywords are site specific.

PLATFORMS

Redhat7, Redhat9, Solaris9

AUTHOR

Paul Anderson <dcspaul@inf.ed.ac.uk >

VERSION

1.1.3-1

B.26 inventory

LCFG inventory component

DESCRIPTION

These **inventory** resources define the inventory information for a cluster of LCFG nodes. There is no **inventory** component; the resources are normally subscribed from a spanning map which is published by **inv** resources on the individual nodes. The special profile format **XMLInventory** can be used to publish an XML copy of the complete inventory from a source file such as /usr/lib/lcfg/source/inventory.

RESOURCES

allocated_*host host!inventory resource*

Inventory resource for *host* (see **lcfg-inv**).

cluster

The name of a spanning map from which the inventory information should be obtained. The default is **inventory/all**.

comment_*host host!inventory resource*

Inventory resource for *host* (see **lcfg-inv**).

date_*host host!inventory resource*

Inventory resource for *host* (see **lcfg-inv**).

domain_*host host!inventory resource*

Inventory resource for *host* (see **lcfg-inv**).

hosts

The list of hosts in the inventory.

location_*host host!inventory resource*

Inventory resource for *host* (see **lcfg-inv**).

maintainer_*host host!inventory resource*

Inventory resource for *host* (see **lcfg-inv**).

manager_*host host!inventory resource*

Inventory resource for *host* (see **lcfg-inv**).

model_*host host!inventory resource*

Inventory resource for *host* (see **lcfg-inv**).

node_*host host!inventory resource*

Inventory resource for *host* (see **lcfg-inv**).

os_*host host!inventory resource*

Inventory resource for *host* (see **lcfg-inv**).

owner_*host host!inventory resource*

Inventory resource for *host* (see **lcfg-inv**).

sno_*host*_*host!inventory resource*

Inventory resource for *host* (see **lcfg-inv**).

tags_*host*_*host!inventory resource*

Inventory resource for *host* (see **lcfg-inv**).

PLATFORMS

Redhat7, Redhat9, Solaris9

AUTHOR

Paul Anderson <dcspaul@inf.ed.ac.uk >

VERSION

1.1.3-1

B.27 ipfilter

Filter rule collection LCFG component

DESCRIPTION

This component collects together the various exporting wishes as expressed in the lcfg and creates a configuration file which will be passed on to the rest of the perimeter filtering mechanism.

RESOURCES

export

domain

Machines ask to export services by adding ”well-known” names to the `export` list. (”Well-known” in the sense that the machines which eventually have to generate the filter rulesets know what they mean!) They can also declare themselves to be in a particular `domain`, for the benefit of the DNS lookup that the filtering host will eventually perform; this will most likely be set as a site default.

defaultDomain

If no domain is set then this is the default to use.

exporting

Some machine somewhere will want to gather together the published export lists. They'll appear as the tag-list `exporting` and corresponding values `export_`*machineName*.

exportexport

exportimport

These two resources form part of the spanning tree glue. Machines which intend exporting services should set the former (it's probably done by default). Machines which collect together the information for passing on should set the latter.

AUTHORS

`George Ross <gdmr@dcs.ed.ac.uk>`

VERSION

0.0.21-1

B.28 iptables

Filter configuration LCFG component

DESCRIPTION

This component configures the iptables network filters.

RESOURCES

prechains

chains

postchains

The `chains` resource specifies which chains we want to add rules to. For each tag in the list, there's a corresponding `rule_`*tag* resource giving the rule to be inserted, or alternatively a `rules_`*tag* resource giving the rule file to be applied. A `policy_`*tag* resource can also optionally be specified; the component checks at configure time whether this is meaningful for the chain in question or not.

The `prechains` and `postchains` resources specify additional chains which should be processed before and after the `chains` chains respectively. It is expected that these will be set as system-wide defaults, rather than for individual machines.

rules

One-off rules can be defined by making an entry in the `rules` list, each tag of which should have a corresponding `rule_`*tag* entry giving the entry to be inserted in the generated script. These rules are then invoked by adding "@tag" to one of the `rules_...` lists above. */

rulesetDir

The final output script is assembled from rules generated by the component itself and rules taken from ruleset files in this list of directories.

configRun

It's sometimes useful to have the configure method automatically run any new rule-file it generates. On the other hand, it's sometimes important **not** to have this happen. Setting this resource causes the file to be run; otherwise it won't be.

inif

outif

If set, define the machine's (external) input and output interfaces respectively.

rsyncFiles

rsyncDir

We may want to rsync in some files first. Which? And where should we put them?

modules

Some kernel modules may have to be loaded first. Which?

mailto

We may want to send a helpful mail message if the rules change. This is where we should send it.

postProcess

This is the name of a postprocessing filter for the assembled rules. It's unlikely that anything other than the default would be appropriate here.

AUTHORS

`George Ross <gdmr@dcs.ed.ac.uk>`

VERSION

0.0.77-1

B.29 irda

The LCFG IrDA component

DESCRIPTION

This component configures the IrDA subsystem.

RESOURCES

tty

The tty to use for the IR port. If this does not have the form ttyS*, it is assumed to be the name of an FIR module.

AUTHORS

```
Alastair Scobie <ascobie@inf.ed.ac.uk>
```

VERSION

0.99.4-1

B.30 kerberos

LCFG Kerberos Component

SYNOPSIS

kerberos *METHOD* [*ARGS*]

DESCRIPTION

An LCFG component that is used to configure and manage the MIT Kerberos service on clients and servers.

METHODS

The non-standard component methods are described below.

propagate

Propagate the current database to this hosts slaves. This method should be regularly called via cron on the master KDC.

buildmaster

Create the master KDC. This method requires input from the user and for reasons of security should only be run directly from the machine's console.

gethostkey

Extract the host key for a given host. This method requires input from the user and as such should not be executed when there isn't a user connected to stdin.

makestash

Create a stash file for a slave KDC. This method requires input from the user and as such should not be executed when there isn't a user connected to stdin.

save

Save the current Kerberos data either as a K5 dump file, a K5 dump patch file, or a tar and gziped copy.

Takes one argument which is the level ranging from 0 to 8. A level 0 is a full copy, level 1 is a diff against the most recent level 0, level 2 is a diff against the most recent level 1, etc. The level argument can also be the string `cp` to take a tar'ed and gzip'ed archive copy instead of a K5 dump. The directory the backups are saved to is specified in the *backup* resource.

K5 dump saves are done using the kdb5_util command and are safe to run live. The tar'ed/gzip'ed copy is not done live and so may be inconsistent if data is changing while it is being made.

This method can only be used on the `master` server. It would normally be invoked automatically at different times and levels via the cron component. All Kerberos save files (whether K5 dump or `cp`) should be kept at the same level of security as the original live data.

load

Reload the current Kerberos data from a save file produced via the save method. Takes one optional argument which is a timestamp filter. With no argument restores to the most recent K5 save. The timestamp argument has the syntax [CC[YY[MM[DD[HH[MM]]]]]]. For example, 200202 would restore to the most recent save for Feb 2002, or 2002021211 would restore to the most recent save for Feb 12 2002 during the 1100 hours period. This method cannot be used to restore from tar/gzip saves.

Invoking this method destroys the existing database and recreates it from the saved data. In some recovery situations you may need to run the *buildmaster* method before doing the load.

This method can only be used on the `master` server.

suspend

Used only on normally disconnected machines (such as laptops) this method will destroy any existing credential cache files in /tmp.

check

Checks whether the root partition is filling up (if it reaches 100% then the KDC continues to respond to authentication requests but with bogus information). If it reaches 90% full this method mails a warning. If it reaches 95% full this method mails a warning and attempts to free up space by deleting older log files. This method can only be used on a master or slave server and would normally be called automatically from cron.

RESOURCES

The non-standard component resources are described below.

GENERAL CONFIGURATION

The following resources control the configuration of clients and servers.

type

Indicates the type of the machine. This can be either `client`, `offline`, `master` or `slave`. Master and slave configure the relevant KDCs, offline indicates that the machine is a client which spends time disconnected, and so shouldn't attempt to do updates when the *start* method is called.

realm

The Kerberos realm that the machine inhabits.

createsasldb

Historical. If set, this will cause the machine to create an empty password database for Cyrus SASL. This was required to allow the GSSAPI SASL mechanism to be used without the application complaining about an empty database, but is uneccessary for newer version of SASL.

CLIENT CONFIGURATION

The following resources control the configuration of clients.

lifetime

Ticket lifetime (also used by the Krb5 PAM module as the renew lifetime).

tktenctypes

Supported encryption types for tickets.

tgsenctypes

Supported encryption types for the ticket granting service.

kdc

The addresses (in the form of *machine*:*port*) of KDCs for the default realm.

randomize

Indicates whether a client should randomize the KDC list before adding it to the configuration file. Use this option with care. Having a KDC other than the master first in this list can cause problems when new services are being installed, as the newly created keys won't be available immediately on the slaves.

admin

The address (in the same form as the kdc address) of the admin server for the default realm.

domain

The default domain of this machine.

domainmap

A space seperated list of domains that should be mapped to this machines default realm.

hostkeyless

If set, disables the creation of a host key for this host. This can be used for lightweight clients, but may have dramatic effects on machines that run Kerberized services, or that require the host key for machine based authentication. Use with extreme care.

checksumtype

Historical. The type (currently numerical) of the checksum to use for mk_safe operations. This was required to make krb5-1.2.1 work correctly.

PAM CONFIGURATION

The following resources (in conjunction with some of the above) control the configuration of the Kerberos PAM service.

forwardable

Set to `true` if the tickets requested by the Kerberos PAM module should be forwardable. Also makes tickets acquired through kinit forwardable if set to `true`.

krb4convert

Set to `true` if the Kerberos PAM module should automaticaly convert Kerberos V tickets to Kerberos IV ones.

maxtimeout

timeoutshift

initialtimeout

Control the timeouts in establishing the connection to the KDC. See the *pam_krb5* manpage for more details.

addressless

Set to `true` if the user should be given addressless tickets, that is ones that can be used from behind a NAT or on a dialup host.

validate

Set to `true` if the user's TGT should be validated against a local service before allowing the user to login. Setting this to `false` opens the machine up to a number of network based attacks.

requiredtgs

Historical. The name of a service who's key is in the local keytab for which the user has to be able to gain a ticket before being allowed to log in. The module's default of `host/` <`hostname` > should serve most needs.

KEY EXTRACTION

The following resources control the automatic creation and extraction of host keys from the KDC to keytabs. This is not the only place that this may occur, individual services may perform their own key extraction.

keys

A list of the keys to extract. These are assumed to be principal names, the actual key extracted will be *key/hostname@default_realm*. If the keytab_*key* resource has no value these will be extracted to the default keytab.

keytab_*key*

The keytab to extract *key* to.

keytabuid_*key*

The UID or username to own the keytab for *key*. Note that if the same keytab is used for multiple keys, then the last key to be extracted will determine the ownership of the keytab. Defaults to root.

keytabgid_*key*

The GID or groupname to own the keytab for *key*. Defaults to root.

SERVER CONFIGURATION

The following resources control the configuration of master and slave KDCs.

slaves

List of FQDNs of machines that slave from this one.

master

List(!) of FQDNs that this machine will accept KDC propagation requests from. There should obviously only be one machine active at propagating at any one time, but this allows for easy recovery from a dead master KDC.

masterkeytype

The type of the KDC master key. Do not change this on a running KDC, unless you are aware of exactly what you are doing.

supenctypes

Encryption types that should be created for keys in the KDC.

kdcenctypes

Encryption types supported for authentication to the KDC.

acls

List of ACL rules for the kadmin server, used as keys for the acl_*tag* resource.

acl_*tag*

Kadmin ACL list entry for *tag*. Together with the *acls* resource, this builds the ACL control file. Entries are as described in the *kadmind(5)* manpage.

krb524d

Whether to run the Kerberos4 compatibility daemon. Default is not to unless this resource has the value `yes`.

directory

If the KDC type is a master and this resource has a value the physical content of */var/kerberos/krb5kdc* is relocated into the given directory and a symbolic link is made from */var/kerberos/krb5kdc* to the new location. This is only ever done once as part of the *buildmaster* method.

kdclog

Location that the KDC should log to.

adminlog

Location that the Admin Server should log to.

backup

Directory where the master servers database backup saves are stored.

mailcheck

mailcheckcc

Email addresses to send fault reports from the *check* method to.

LOCAL AUTHENTICATION

The following resources control the configuration of local authentication for operation when disconnected (or no route to KDC).

rootpwd

A lauth crypted string containing the root password for the machine. This will probably eventually go away, in favour of extracting this directly from the KDC.

localusers

A space seperated list of those users who are allowed to log in to this machine when it is disconnected. This is used both on the client (to decide whether to extract keys) and on the key server (via an LCFG spanning map).

FILES

/etc/krb5.conf

/etc/krb5.keytab

/var/kerberos/krb5kdc/kdc.conf

/var/kerberos/krb5kdc/kadm5.acl

/var/kerberos/krb5kdc/kpropd.acl

/etc/localpasswd

/etc/localusers.conf

PLATFORMS

Redhat7 Redhat9

SEE ALSO

kdb5_util, kadmin.local, kprop, pwdclient, kdcpwdserver

AUTHOR

DICE Authentication and Authorization Team <auth-team@inf.ed.ac.uk >

VERSION

1.32.22-1

B.31 kernel

LCFG kernel component

DESCRIPTION

This component configures kernel parameters via the `/etc/sysctl.conf` file. Any changes from the existing file cause a reboot to take place. It also builds /boot/vmlinuz and /boot/initrd.img links, if required. It will also rebuild kernel modules when certain rpms have been updated.

set

A list of variables to be set. Each entry requires an associated tag and value. Required.

tag_*X*

The name of the kernel variable to be set. Required.

value_*X*

The value to be assigned to X. Required.

mkkernellink

If this resource is non-null, the component will create a link `/boot/vmlinuz` to the current kernel.

mkinitrdlink

If this resource is non-null, the component will create a link `/boot/initrd.img` to the current initrd image.

kerneltype

Defines what version of the kernel to use. Default null value indicates the uniprocessor kernel. Set to SMP for the SMP kernel and bigmem for a bigmem kernel. Note, this assumes standard Redhat kernel naming conventions.

srcmodules

A list of modules to rebuild when certain rpms (eg the kernel) are updated. Typically these are kernel modules.

triggers_*module*

A list of rpms that trigger a rebuild of the specified module if any of them are upgraded/removed etc.

script_*module*

The filename of the script to build the specified module. It is called with a parameter of `install` when the module is being rebuilt, or `remove` if the module has been removed from the `srcmodules` resource. If no script is specified, the component will assume the script is called /usr/lib/lcfg/conf/scripts/*module*.

AUTHORS

`Alastair Scobie <ascobie@inf.ed.ac.uk>`

VERSION

0.101.6-1

B.32 ldap

LCFG LDAP Component

SYNOPSIS

ldap *METHOD* [*ARGS*]

DESCRIPTION

An LCFG component that is used to configure and manage the OpenLDAP service on clients and servers.

METHODS

The non-standard component methods are described below.

kick

Force the LDAP server to replicate from its master. By default this does not delete any entries that have been deleted on the master.

Supplying the `hard` argument will cause deletions to be performed too. Deletions may take a considerable amount of time, and significantly increase the load on the LDAP master.

autokick

Normally called from a crontab entry. The intention is to trigger a normal kick hourly (at a random number of minutes past the hour based on host ip) and do a hard kick daily (at a random hour based on host ip). However, arguments to this method allow the timing of normal and hard kicks to be adjusted for more or less frequent replication. The way this method works is unfortunately rather tied to the way that the LCFG cron component works. For the default behaviour there should be a crontab entry set to call this method hourly at the same exact random number of minutes past the hour (this can be achieved via the LCFG cron component by using an AUTO value for minutes since this calculates a random number from an IP in the exact same way as this component).

This method takes two optional arguments, a fixed hour (or AUTO) and a fixed number of minutes past the hour (or AUTO). For example to run the hard kick at 2pm (overriding the randomly chosen hour) do `autokick 14`; to run the normal and hard kicks at 5mins past the hour (overriding the randomly chosen minutes past the hour) do `autokick AUTO 5`; to set both values you can do `autokick 14 5` (so run the normal kick hourly at five minutes past the hour and the hard kick daily at 14:05). Note that compatible adjustments must also be made to the crontab entries. To have the normal kick every two hours instead of every hour but keep the hard kick daily you would call this method with the args `14 AUTO` and have `AUTO */2 * * * *` as the crontab entry (where `AUTO` in both cases is replaced by the same number if not using the LCFG cron component).

We really need a better way to do all this, possibly by improvements to the LCFG cron component, or by getting ldapreplicate to handle the automatic replication by daemonizing and controlling the timing without using cron.

rebuild

Force the LDAP server to delete all of its data and re-replicate from the master. This method can only be used on a slave server. Useful if database corruption is suspected.

save

Save the current LDAP data as an LDIF file or LDIF patch file. Takes one argument which is the level which can be from 0 to 8. A level 0 is a full copy, a level 1 is a diff against the most recent level 0, a level 2 is a diff against the most recent level 1 etc. The level argument can also be the string `cp` to take a tar'ed and gzip'ed archive copy instead of an LDIF dump. The directory these files are saved into is specified in the *backup* resource.

Saves are currently done live (without stopping slapd) so may contain inconsistency if the data was being updated at the same time as the save.

This method can only be used on the master server and would normally be called automatically at different times and levels via the cron component.

load

Reload the current LDAP data from a save file produced via the save method. Takes one optional argument which is a timestamp filter, without this argument the most recent LDIF save will be restored.

The timestamp argument has the syntax [CC[YY[MM[DD[HH[MM]]]]]]. For example, 200202 would restore to the most recent save for Feb 2002, or 2002021211 would restore to the most recent save for Feb 12 2002 during the 1100 hours period. This method cannot be used to restore from tar/gzip saves.

This method stops the slapd process, deletes any current data, restores data and restarts the slapd process. The LDAP directory will be unavailable whilst the load is being carried out.This method can only be used on the master server.

check

Checks whether the slapd process has stopped running when the component status indicates that it should be running (this would be the case if it has crashed for some reason). If so it restarts it and mails a fault report. This method can only be used on a master server and would normally be called automatically from cron.

RESOURCES

The non-standard component resources are described below.

CLIENT CONFIGURATION

The following resources control the configuration of clients.

These items configure the client's default LDAP server. The default is not universally used, in particular only those tools built on the OpenLDAP C libraries will pay attention to this section of configuration.

server

The address of the LDAP server the machine should query.

searchbase

The base DN for searches on that server.

ldapversion

The LDAP version to use for queries.

binddn

The DN to bind to the server as (uses an anonymous bind if this is omitted).

bindpw

The password to use if the bind is not anonymous, and requires a password.

SERVER CONFIGURATION

The following resources control the configuration of servers.

type

Type selects which mode the LDAP server is running in:

master

LDAP server is domain master. A minimal initial dataset is loaded from the file given in the *initialldif* resource. No other data is loaded.

slave

LDAP server is a slave. The initial dataset is loaded by means of an ldapsearch from the domain master. Further replication is determined by the contents of the *replmethod* resource.

client

No LDAP server is run.

MASTER SERVER CONFIGURATION

The following resources are specific to the configuration of the master server.

initialldif

The name of a file in /usr/lib/lcfg/conf/ldap holding a minimal initial data set in LDIF format to bootstrap the master server. Default is *root.ldif*.

backup

Base directory where the master servers database backup saves are stored. They are stored in the `new` sub-directory of this which holds the most current backup files and is kept to a certain size controlled by the *backupmax* and *backupmin* resources and the `old` sub-directory where older backups are kept forever (they must be manually cleared if filespace is needed). Generally the `new` sub-directory would be expected to be backed up onto an offline medium and/or mirrored onto another machine.

backupmax

Maximum size of the `new` backup directory (in 1K blocks). When this size is exceeded (on doing a backup) it triggers a move of files out of the `new` backup directory and into the `old` backup directory. This file move continues until the size of the `new` backup directory falls below the value of the *backupmin* resource (see below).

backupmin

Minimum size of the `new` backup directory (in 1K blocks). Files are only moved out of the `new` backup directory until this minimum size is reached. Depending on the average backup size and frequency and level configuration of backups this determines the scope of the `new` backup directory, ie. the time period backups in `new` cover.

mailcheck

mailcheckcc

Email addresses to send fault reports from the *check* method to.

GENERAL SERVER CONFIGURATION

The following resources are for the configuration of all servers.

directory

The directory in which the LDAP database is stored. Defaults to /var/lib/ldap

configtemplate

Name of the file in /usr/lib/lcfg/conf/ldap which is the slapd.conf template for pre-processing by *sxprof(8)*. Defaults to *slapd.conf.tmpl*.

logfacility

Name of the syslog facility to which logging should be performed.

loglevel

The level at which logging should be performed. The slapd.conf(5) manpage provides details of what information is provided at each level.

ldapschemas

List of schemas to include in the slapd configuration. If the schemafile_*TAG* resource is present this contains the name of the file to use, otherwise it defaults to */etc/openldap/schema/TAG.schema*.

schemafile_*TAG*

Filename of schema file to use for *TAG*.

writemaster

The host that LDAP update requests on a slave server should be referred to using the DN in the *dbrootdn* resource to make the actual update. Should be empty on the master server.

allowv2

Set to a non-null value to allow LDAP v2 binds.

aclfile

The name of the file in /usr/lib/lcfg/conf/ldap containing ACLs for the directory service.

changelogdn

If present, turns on in-directory changelogs, storing them in the location given. Changelogs are needed for internal trigger support.

dbsuffix

Naming suffix of the database that the LDAP server stores. Will generally be the same as *searchbase*.

dbtype

Type of backend database. Generally `bdb` or `ldbm`.

dbrootdn

RootDN of the database. On a slave, this should be the DN used by the replication agent which copies content into the database, on the master it should be the DN which has 'super user' access to the database, or a non-existent DN to disable this form of access.

indices

List of attributes which should be indexed. Note that changing this list will trigger a database shut down and index rebuild. Depending on the complexity this may take a large amount of time.

indextype_*TAG*

List of the indices to maintain for attribute *TAG*. See the slapd.conf(5) manpage for more details.

sizelimit

timelimit

idletimeout

See the slapd.conf(5) manpage for details.

lastmod

checkpoint

See the slapd-bdb(5) manpage for details.

bdb_cachesize

bdb_lg_regionmax

bdb_lg_bsize

bdb_lg_dir

These backend specific resources set the corresponding parameters in the main database's DB_CONFIG configuration file. See the BDB documentation for more details.

saslrealm

The default realm for all SASL operations against the server

REPLICATION SERVER CONFIGURATION (slurpd)

The following resources control how the server makes its information available to replication agents using slurpd.

slurp

If `yes` manage the starting and stopping of the slurpd process.

slaves

List of FQDNs of machines that are slurpd slaves of this one.

replicaconf

List of additional configuration to add to the replica line for each slurpd slave.

REPLICATION CLIENT CONFIGURATION

These resources control how a slave server replicates from another server (normally the master server).

replicatype

The type of replication in use. Currently the only supported option is `ldapreplicate`.

master

Name of the server to fetch the initial LDAP configuration from. This doesn't have to be the master LDAP server for the domain. When the server type is a `slave` this controls the server that the machine is replicated from.

FILES

/etc/ldap.conf

/etc/openldap/ldap.conf

/etc/openldap/slapd.conf

*/etc/openldap/schema/**

PLATFORMS

Redhat9

SEE ALSO

ldapreplicate, slapd, slurpd, slapadd, slapcat, slapindex

AUTHOR

DICE Directory Service Team <dirservices-team@inf.ed.ac.uk >

VERSION

2.0.28-1

B.33 localhome

LCFG localhome component

DESCRIPTION

This objects builds the local home directories for those users listed in the `users` resource. It also builds an automount map for these directories, with a redirect (`redirect` resource) for users who aren't listed in the `users` list.

RESOURCES

users

Specifies which users should have local home directories. Groups of users can be added by prefixing a netgroup name with an @ symbol.

virtual

The virtual name of the directory containing the local home directories. Defaults to `/localhome`. This name links to the directory specified by the `physical` resource.

physical

Specifes the real directory in which the local home directories should be created.

redirect

Used when generating the automount map to specify the default destination for users not listed in the `users` resource.

mapfile

The filename of the generated automount map. If empty it will default to `/var/lcfg/conf/amd.localhome.map`.

maptype

The type of automounter map which should be created. Defaults to `amd`.

grouphelper

The shell command that will take the name of a group as an argument and return a list of users in that group. For each group mentioned in the `users` resource the characters '%s' will be replaced by the name of that group. An example command might be

```
/usr/bin/netgroup -U %s
```

PLATFORMS

Redhat7, Redhat9

AUTHOR

Alastair Scobie <ascobie@inf.ed.ac.uk>, Ken Dawson <ktd@inf.ed.ac.uk>

VERSION

2.0.9-1

B.34 logserver

LCFG logserver

DESCRIPTION

This component serves log files and other information about LCFG components via HTTP. The HTTP server listens on port **lcfglog** (default 734), and provides information on the following URL pathnames:

profile/*component*.html

The resource values from the current profile in HTML.

profile/*component*.txt

The resource values from the current profile as a text file.

profile/long/*component*.html

The resource values from the current profile with full details, as an HTML file.

status/*component*.html

The resource values from the current status in HTML.

status/*component*.txt

The resource values from the current status as a text file.

status/long/*component*.html

The resource values from the current status with full details, as an HTML file.

log/*component*.html

The current log file in HTML.

log/*component*.txt

The current log file as a text file.

err/*component*.html

The current error file in HTML.

err/*component*.txt

The current error file as a text file.

warn/*component*.html

The current warning file in HTML.

warn/*component*.txt

The current warning file as a text file.

doc/*component*.html

The documentation in HTML.

doc/*component*.pod

The documentation in pod format.

RESOURCES

block

A space-separate list of components whose log files should not be published. Typically used on servers to prevent publication of sensitive log files such as authorization.

components

The list of components for which logfiles should be published (unless specified in the block list).

logrequests

True to log all requests.

maxlines

The maximum number of lines to display in one HTML log page (default 500).

statusurl

The root of the URL used to access the server status page for this client. The domain name and hostname are appended to this base to create the link to the status page (default http://lcfg/status).

PLATFORMS

Redhat7, Redhat9, Solaris9

AUTHOR

Paul Anderson <dcspaul@inf.ed.ac.uk >

VERSION

1.1.12-1

B.35 lprng

The lcfg lprng component

DESCRIPTION

Component to start, stop and configure the lprng lpd daemon.

RESOURCES

printers

A list of print queues spooled from this server.

debug

The numeric debug value to pass to lpd with the `-D` flag.

owner

The user/uid that the lpd daemon runs as.

kerbprinc

A boolean value indicating whether this machine requires an lpr kerberos principal.

localname

Local printing only. Name of queue for local printer (will default to 'local' but can be overridden in by user-maintained profile), e.g. lprng.localname[localprinter=myprinter] myname

localsendto

Local printing only. Specifies exactly where the print job is to be sent (e.g. /dev/lp0 for a locally attached parallel printer, /dev/usb/lp0 for USB, or smbprinter@smbhost.somewhere.org for a networked printer). This resource essentially equates to the lp= bit of the printcap entry.

localformat

Local printing only. Output format of the printer - formats currently supported are Postscript and Ghostscript. The latter refers to any format that will be converted to from Postscript, by gs.

localopts

Local printing only. Additional colon-separated options needed by printing system. The options supported here depend on the format of the printer (e.g. For a Postscript printer, we need to know the ppd file to use, for an Inkjet we might need colour options, or the 'device' name) and need to be supported by the backend. It could be something along the lines of (for a Deskjet) "device=deskjet,colour=CMYK"

localpcap

Provided to override printcap options if required. e.g. if it was set to if=/path/toby/my/filter:sh:sf then those printcap entries only would be overridden.

METHODS

start

stop

configure - The configure method performs the following steps - (1) checks that the appropriate spool directories are in place, with the appropriate permissions and creates/modifies them if not; (2) performs any necessary steps relating to the creation and registration of an lpr server principal.

NOTES

This component assumes that the rest of LPRng is installed and configured appropriately. When adding a new printer, the queuename should be added to the **printers** resource as the final step - printcap information must be in place prior to this.

AUTHORS

```
Toby Blake <toby@inf.ed.ac.uk>
```

VERSION

0.99.38-1

B.36 mailng

LCFG mail component.

DESCRIPTION

This component configures the sendmail (client only) service.

RESOURCES

aliasfile

The full pathname of the AliasFile parameter for the sendmail.cf file.

daemon

If this option if non-null, then the sendmail daemon will listen on the SMTP port for connections. The `run_daemon` option must be set for this option to be useful.

daemonportoptions

Normally used by the sendmail.cf template to set the DaemonPortOptions field. The default is to only accept connections from localhost.

local

Normally used by the sendmail.cf template to set the DH field.

mctmpl

If this resource is present, the sendmail.cf template (smtmpl) is created by first passing this file through `sxprof` and then m4.

mode

Normally used by the sendmail.cf template to set the delivery_mode (default "background"). Set this to "q" to have mail dumped in the queue rather than being delivered immediately. Useful for portables when talking to SMTP servers that are very slow to respond.

poll

If this option is set, then the sendmail daemon will poll the mail queue at the specified intervals (default "1h"). You probably want to set this to some small value (30s?) when using queued delivery mode. The `run_daemon` option must be non-null for this option to be useful.

relay

Normally used by the sendmail.cf template to set the DS field.

run_daemon

Set this non-null to run a sendmail daemon. This is required if either the `daemon` or `poll` options are set.

smtmpl

The sendmail.cf template. The sendmail.cf file is created by passing this template through `sxprof` with all mailng resources defined. If this is null, then the sendmail.cf is not changed.

smconfig

Where to put the processed sendmail template file. If this is null, then a warning is logged and no sendmail.cf file is produced.

Dealing with root mail

rootmail

A space separated list of email addresses that root mail should be copied to. This only has an affect if the mail server that actually receives root mail is running the `rootredirect` script.

cluster

The identifying spanning map cluster that this rootmail belongs to.

SEE ALSO

rootredirect(8)

AUTHORS

```
Paul Anderson <paul@dcs.ed.ac.uk>
Neil Brown <neilb@inf.ed.ac.uk>
```

VERSION

1.7.3-1

B.37 network

LCFG network component

DESCRIPTION

This component configures the /etc/sysconfig/network-scripts configuration files and /etc/hosts.

offline

This resource, if set to yes, stops the object from making any configuration changes when the start method is invoked. This is handy for portables where the run method is user invoked to make configuration changes.

interfaces

A list of ethernet interface names. Each interface must have the following tagged resources.

hostname_*interface*

This resource specifies the hostname for this interface.

device_*interface*

This resource specifies the ether device for this interface. The default value of auto indicates that the ether device is set to the tag key *interface*.

ipaddr_*interface*

This resource specifies the IP address for this interface. The default value of auto indicates that the component should attempt to resolve the IP address.

netmask_*interface*

This resource specifies the netmask for this interface.

network_*interface*

This resource specifies the network for this interface. The default value of auto indicates that the component should derive the network from the IP address. A class C address is currently assumed.

broadcast_*interface*

This resource specifies the broadcast address for this interface. The default value of auto indicates that the component should derive the broadcast address from the IP address. A class C address is currently assumed.

onboot_*interface*

This resource specifies whether this interface should be configured at boot time. The default value is ”yes”.

bringup_*interface*

This resource specifies whether this interface should be brought up manually by the network component at start time. This is useful for interfaces that aren’t prepared at system boot time (eg VLANs). The default value is ”no”.

hostsorder_*interface*

This resource specifies which form of hostname is entered into the /etc/hosts file. The value full specifies that the fully qualified name should be entered, while the value short specifies that just the simple hostname should be entered. Both values can be specified, with order being significant.

extrahosts

A list of additional entries for the /etc/hosts file. Each entry should have the form hentry_*tag*.

hentry_*tag*

The value for the /etc/hosts entry denoted by *tag*.

gateway

A list of gateways for this machine. The actual gateway used will be chosen randomly from this list.

gatewaydev

The ethernet interface to use to communicate to the default gateway.

hostschangereboot

This resource specifies whether changes to `/etc/hosts` should trigger a reboot. The default value is ”yes”.

AUTHORS

Alastair Scobie <ajs@dcs.ed.ac.uk>

VERSION

1.99.8-1

B.38 nfs

LCFG nfs component

DESCRIPTION

This object configures the NFS service. It creates the list of exported filesystems and their mount options and saves them in the relevant exports file.

NB: it does not start NFS - this is done outwith the lcfg system.

exports

A list of filesystems to export.

fs_*fsys*

The pathname of the filesystem to export for this tag.

fs_*foptions*

The mount options for the named filesystem.

FILES

/etc/exports

AUTHORS

`Alastair Scobie <ascobie@inf.ed.ac.uk>, Jeremy Olsen <J.Olsen@ed.ac.uk>`

VERSION

1.0.2-1

B.39 ngeneric

LCFG new generic component.

DESCRIPTION

This component is intended for inclusion by other LCFG components. It provides a supporting framework including default methods and utility functions.

The components should include **/usr/lib/lcfg/components/ngeneric** and call the **Dispatch** function with the command line arguments. The lcfg component **example** shows how this is used in practice.

FUNCTIONS

Components can override the following functions:

Configure

This routine is called when the **configure** method, is invoked, as well as **start** and **restart**. **ngeneric** will have placed the values of all resources into the environment with variable names of the form `LCFG_`*resource*.

This routine should (re)create any necessary configuration files, and restart or signal any affected daemons. It is up to the component to determine which (if any) individual resources have changed and to minimize the reconfiguration appropriately (the template processor can help with this).

The component should call Fail() if the reconfiguration fails.

Start

This routine gets called when the **start** or **restart** methods are invoked, either manually, or at boot time. **Configure** will be called before **Start** leaving the resources available in the environment.

The component should override this routine to start any necessary daemoms.

Stop

This routine gets called when a component is stopped, either manually, or at shutdown time (or for a restart). The component should override this routine to stop any necessary daemons. When the routine is called, ngeneric will have placed the configuration (as saved at the last configure) into the environment.

Run

This routine gets called when the **run** method is invoked. The component should override this routine to perform any necessary operations. When the routine is called, **ngeneric** will have placed the configuration (as saved at the last configure) into the environment.

LogRotate

This routine gets called when the **logrotate** method is invoked, normally by the logrotate script when the logfile has been rotated. The component should override this routine to arrange for any running daemons to release the logfile. The environment contains the configuration saved at the last configure/start.

Suspend

Take any APM suspend action. The environment contains the configuration saved at the last configure/start. This routine is not protected by the normal semaphore.

Resume

Take any APM suspend action. The environment contains the configuration saved at the last configure/start. This routine is not protected by the normal semaphore.

Reset

Reset the error and warning status files. The existence of these files determines the status of the error and warning icons on the server status page. These files are deleted when the component starts.

Status

Display status information. The environment contains the configuration saved at the last configure/start. The default routine displays the values of the resources at the last configuration.

Log

Display log information. The environment contains the configuration saved at the last configure/start. The default routine displays the current logfile.

Monitor

Report monitoring information (by calling **Notify**) for the tag specified by the first argument. The environment contains the configuration saved at the last configure/start. The default routine reports an error.

INPUT/OUTPUT

Components should avoid writing to STDOUT/STDERR since this may be lost, or may clutter the startup screen. The functions Debug(), Info(), Warn() and Fail() should be used to output short messages. By default, STDERR and STDOUT are redirected to the component logfile. The file descriptors 11 and 12 are opened on the original STDOUT and STDERR respectively for those cases where a method absolutely needs to write to these channels - for example to print a console prompt, or to perform a Log() or Status() method.

RESOURCES

Some component resources are interpreted by the ngeneric component or the LCFG client. The names of these resources all begin with **ng_** and care should be taken not to use these names for other purposes:

ng_cfdepend

This resource is interpreted by the LCFG client 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. The default is $<$*self*.

ng_cforder

The client default file specifies that the server should use this resource to order the **client.components** list. (The **client.components** resource specifies the order in which components should be reconfigured after a configuration change.) **ng_cforder** specifies a list of constraints on the the order in which the components are reconfigured. A constraint of the form $>$*comp* means that this component must be configured after *comp*. Similarly, $<$*comp* means that this component must be configured before *comp*.

ng_debug

Set the **_DEBUG** variable.

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_*tagtag!ngeneric resource*

The logrotate line corresponding to *tag*.

ng_monitor

If this facility is set to the name of a file, then all errors, warnings and monitoring information will be appended to the named file, if it exists. This is typically set to a named pipe (eg. `/var/lcfg/tmp/monitor.fifo` to transmit information to a monitoring system (eg **lcfg-pemsensor**).

ng_prod

If this resource changes, the client will call the method specified by **ng_prodmethod** instead of calling **ng_reconfig**. The value of the resource is specifically unused. This can be used to force one-off execution of a particular method. For example, by setting **ng_prod** to some new value (typically a timestamp) and **ng_prodmethod** to **restart**, the component will restart when the new profile is received.

ng_prodmethod

The component method to call to ”prod” the component.

ng_reconfig

This resource is interpreted by the LCFG client to determine the method to call when the component resources have changed.

ng_statusdisplay

If this resource is **true** then the component will appear in the server status display. (default is true). If it has the value **nocomp**, then the component is assumed to be a ”pseudo component” with no corresponding running code - in this case, no client acknowledgements are expected, and the component shows as ”ok”, rather than ”unknown”.

ng_syslog

If this variable is set to the name of a **syslog** facility (eg. **local3**), then all error and warning messages will be copied to syslog with the specified facility.

ng_verbose

Set the **_VERBOSE** variable.

LOCKING

ngeneric uses **lcfglock** to create a semaphore on all method calls (with the exception of those noted above). The method **unlock** can be used to force the removal of the lockfile.

LOG ROTATING

When the **configure** method is called, **ngeneric** will look for a logrotate configuration file in `/usr/lib/lcfg/conf/`*component*`/logrotate`. This is passed through the template proprocessor **sxprof** to allow per-machine configuration.

If the component does not provide a logrotate file, the ngeneric logrotate file is used. This rotates the component logfile at some default interval and calls the component **logrotate** method in the postrotate script. The default logrotate file allows extra parameters to be added directly from component resources using the `logrotate` resource. Eg:

```
foo.logrotate a b
foo.logrotate_a    daily
foo.logrotate_b    rotate 7
```

VARIABLES

ngeneric creates local shell variables beginning with ”_”. The following variables may be of general use:

_COMP

The name of the current component.

_DEBUG

Enables debugging information. Set by a **-D** option.

_DUMMY

Normally used to perform a dummy execution of the method call for testing. Set by a **-d** option.

_LOGFILE

The name of the log file.

_NOSTRICT

Method specific flag, normally used to force a less strict interpretation of method semantics. For example, the start method will exit silently if the component is already running, rather than fail.

_OKMSG

The message to be displayed when the method completes sucessfully. Components may append a string of the form {\tt (}{\em message}{\tt)} to this variable, to display additional status information when the method exits.

_QUIET

Disables unnecessary messages, including the "OK" message. This is useful when calling components from cron. Set by a **-q** option.

_STATUSFILE

The name of the status file.

_TIMEOUT

The lock timeout (in seconds). Set by a **-t** option.

_VERBOSE

Enables additional informational messages. Set by **-v** option.

AD-HOC METHODS

It is possible to create additional ad-hoc methods. These should be exported with names of the form `Method_`*methodname*, and they will be automatically called by the Dispatch function. Ad-hoc methods should arrange to call the **Lock** function if appropriate to prevent simultaneous method calls.

SEE ALSO

lcfg-example

An example component.

sxprof

The template processor.

AUTHORS

```
Paul Anderson <dcspaul@inf.ed.ac.uk>
```

VERSION

1.1.23-1

B.40 nscd

LCFG NSCD Component

SYNOPSIS

nscd *METHOD* [*ARGS*]

DESCRIPTION

An LCFG component that is used to configure and manage NSCD, the Name Service Cache Daemon.

METHODS

The component only has standard methods.

RESOURCES

The non-standard component resources are described below.

threads

The number of threads that the daemon should use (optional).

maps

A space seperated list of the maps that the daemon should manage. Currently only `passwd`, `group` and `hosts` are supported by NSCD.

positivettl_*map*

The time-to-live for successful matches in the given map in seconds.

negativettl_*map*

The length of time to cache failed lookups in the given map for in seconds.

suggestedsize_*map*

The suggested size of the cache for a given map. This should be a prime number.

checkfiles_*map*

Whether to check the standard file for a given map to determine whether to invalidate all cache entries. Should be either `yes` or `no`.

FILES

/etc/nscd.conf

PLATFORMS

Redhat9

SEE ALSO

nscd

AUTHOR

DICE Directory Service Team <dirservices-team@inf.ed.ac.uk>

VERSION

1.5.5-1

B.41 nsswitch

LCFG nsswitch component

DESCRIPTION

This object constructs an nsswitch.conf file from information in the LCFG database.

maps

A list of the maps which should be included in the nsswitch.conf file

mods_*map*

A list of modules for each map. These are nsswitch modules such as "files" "nis" "ldap" and the like. The ordering should be as required in the file.

AUTHORS

```
Alastair Scobie <ajs@dcs.ed.ac.uk>
```

VERSION

0.100.6-1

B.42 ntp

The LCFG NTP component

DESCRIPTION

This object constructs all the necessary configuration files and starts the `ntp` time daemon.

The `run` method will run ntpdate and set the hardware clock, provided that the `run_daemon` resource was not set and so no daemon was started. This is useful on laptops, where a network connection may be unavailable or it may be undesirable to bring one up.

RESOURCES

run_daemon

This resource should be set to enable the `ntp` daemon. It should normally be set to on permanently connected machines and not set on normally disconnected machines (e.g. laptops). If no daemon is started then the component's `run` method can be used to resynchronise the machine's clock.

servers

A (space-separated) list of NTP servers.

peers

A (space-separated) list of NTP peers.

restrict_default

restrict_policy

restrict_localhost

Access restrictions to apply. `restrict_default` specifies what the global default is. `restrict_policy` allows for separate site-specific restrictions to be applied. `restrict_localhost` specifies what restrictions to apply to other things running on the machine itself. The component itself will inject its own appropriate restrictions for any configured servers and peers.

NOTE: if you don't specify a value for `restrict_default` and `restrict_localhost` then those restrictions are turned off. If no value is specified for `restrict_policy` then no restriction or unrestriction statement is generated in the daemon configuration.

minpoll

maxpoll

These two set the minpoll and maxpoll values for all the configured servers and peers. See the documentation in the ntp distribution for full details. The daemon will operate quite happily without these being set, so if in doubt leave them alone.

contextlabel

This resource does not actually affect the operation of the component, but instead is included in some of its messages. Setting it to some lcfg context-specific value might therefore be useful to the user.

configfile

The name of the daemon's (generated) configuration file.

driftfile

The name of the daemon's drift file.

pidfile

The name of the file into which the daemon should write its pid.

ntpd

The name of the daemon program.

ntpd_flags

Additional command-line flags to pass to the daemon.

ntpdate

The name of the `ntpdate` program.

tickadj

The name of the `tickadj` program.

raiseprio

Should the component attempt to raise the daemon's priority, so that other processes interfere less with time-keeping?

logconfig

Should the daemon do any logging? The value of this resource should be a list of valid logconfig keywords. If it's not set then no logging is done. See the standard NTP web documentation for details of what's required here.

statistics

statsdir

filegen_...

Should the daemon collect statistics? `statistics` says which we should collect, if any. `statsdir` says where they should go. `filegen_thing` which says how that particular statistic is to be handled. See the standard NTP web documentation for details of what's required here.

monitor

Should the daemon keep track of protocol requests, to be queried using `ntpq`'s `monlist` command?

getaddr

A helper program used by the component itself. Do not set this resource unless you know what you're doing.

PLATFORMS

RedHat 7, RedHat 9. Previous versions ran on Solaris 2.6.

AUTHORS

```
George Ross <gdmr@inf.ed.ac.uk>
```

VERSION

2.1.13-1

B.43 pcmcia

LCFG pcmcia component

DESCRIPTION

This object configures and starts the PCMCIA system.

pcic

The type of PCMCIA controller chip for this machine.

pcic_opts

Loadtime options for the PCMCIA controller kernel module.

core_opts

Loadtime options for the PCMCIA core kernel module.

cardmgr_opts

Loadtime options for the PCMCIA `cardmgr` daemon.

config_opts

A list of line tags for the `/etc/pcmcia/config.opts` file.

conf_*tag*

The `/etc/pcmcia/config.opts` line associated with *tag*.

suspend_restart

If set to `yes`, the pcmcia service will be stopped on suspend and (re)started on resume.

AUTHORS

```
Alastair Scobie <ascobie@inf.ed.ac.uk>
```

VERSION

0.100.2-1

B.44 perlex

An example LCFG component in Perl

DESCRIPTION

This component is an example only.

RESOURCES

server

An example resource which gets substituted into the configuration file.

PLATFORMS

Redhat7, Redhat9, Solaris9

AUTHOR

Paul Anderson <dcspaul@inf.ed.ac.uk >

VERSION

1.1.3-1

B.45 profile

Client resources used by the LCFG server

DESCRIPTION

There is no LCFG **profile** component, but **profile** resources for an LCFG client are used by **mkxprof** when compiling a client profile. Every client whose profile is to be generated by **mkxprof** (**lcfg-server**) must have a set of **profile** resources.

RESOURCES

acl_*access_fileaccess_file!profile resource*

An Apache "**allow from**" specification for the access file corresponding to *access_file*, specifying hosts which should be permitted to access this profile without authentication.

auth

A list of tags representing Apache web access files to be created in the profile directory for this host.

authorize

The name of a Perl module to use for default authorization (for example by **om**). This is not used by the client, or the server; it simply provides a common source of reference for other components. The default is **LCFG::Authorize**.

comment

A comment for inclusion on the server status page.

components

A space-separated list of components whose resources are to be included in the profile. mkxprof wil not generate profiles for components not specified in this resource.

domain

The client domain. This defaults to the same domain as the server.

file_*access_fileaccess_file!profile resource*

The name of the Apache access file for the given *access_file*. Normally, this will be **.htaccess**, but Apache may be configured to use multiple different access files in different circumstances: for example, a different access file may be used for SSL and plain HTTP.

format

This resource specifies the name of the Perl module used to generate the profile. This can be used to generate profiles in different formats. The only format currently supported (and the default) is XML.

group

An (optional) three digit numeric order number, followed by a title string. Hosts are grouped by **group** in the server status display. The title string is the title for the display section and the order number is used to sort the sections. By default, the group is set to the domain, and the order number is 100.

maxupdate

The maximum time expected between package updates on the client. A warning icon will be displayed on the status page, if the client has not performed a successful package update within the last **maxupdate**. The value of the resource should be an integer, followed by **h** (hours).

node

The client node name (host name). This defaults to the name of the source file.

notify

If this resource is true, a UDP notification wil be sent to the node whenever the profile changes.

packages

A list of package specifications (eg. RPMs) to be included in the profile. Each specification may be either:

A ”package list” file

The specification is the name of the file, proceeded by **@**. The file should contain a list of package names of the form *name-version-release*{**:***options*}, optionally proceeded by **+** or **-**.

A package name

Of the form *name-version-release*{**:***options*}, optionally proceeded by **+** or **-**.

A tag name

This is tag for a resource name of the form **profile.packages**_*tag* which is assumed to contain further package specifications (these can be nested to an arbitrary depth).

Package specifications occuring either in a resource value, or in a file may be followed by a context specifier in the usual form. Context specifications are not permitted on the tagged resources, or on filenames.

passwd

The passwd for web access. This is entered into the password database which is referenced by the access control file to protect the directory containing the profile. This resource is cached separately by the client so that it is only available to root processes. The value is returned as ”****” by the client libraries if it set, and null otherwise.

pwf_*access_fileaccess_file!profile resource*

The name of an Apache DB password file which will be used to to authenticate profile requests when the access control conditions are not satisified (see **acl**_*access_file*), or not present. The client will attempt connections using the FQDN of the host as the username, and the value of the **profile.passwd** resource as the password. The special value **auto** can be supplied in which case the server will use the password file created automatically from the **profile.passwd** resources.

release

The configuration release. This value will be substituted for the string **%r** in the pathnames of any directories included by the server. If this is null, the string **default** will be substituted.

rungroup

The groupname under which the LCFG system runs by default on the client.

runuser

The username under which the LCFG system runs by default on the client.

softrelease

The expected software release version on the client. If this resource is present, then a warning icon will be displayed unless the release string matches the contents of /etc/LCFG-RELEASE.

version_*componentcomponent!profile resource*

This resource specifies the version of the **.def** file to be used for the specified component. The file is named *component-version***.def** or *component***.def** (if there is no version specified).

AUTHOR

Paul Anderson <dcspaul@inf.ed.ac.uk >

VERSION

2.1.64-1

B.46 ramdisk

LCFG ramdisk component

DESCRIPTION

This object creates and configures one or more ramdisks.

RESOURCES

disks

A list of digits specifying the ramdisks to create. The disks will be mounted on /ramdisk/ <N >.

size_*NN!ramdisk resource*

The size (in K) of ramdisk <N >.

users_*NN!ramdisk resource*

A list of users who will have directories created a ramdisk <N >. Each directory will be named after, and owned by, the corresponding user. Permissions are set to 0700.

PLATFORMS

Redhat7, Redhat9

AUTHOR

Paul Anderson <dcspaul@inf.ed.ac.uk >

VERSION

1.3.0-1

B.47 rmirror

An LCFG component for offline disc mirroring.

DESCRIPTION

This object carries out offline mirroring by invoking rsync.

RESOURCES

disklist

A list of disks to be rsynced to the local machine.

srchost_*tagtag!rmirror resource*

The host serving the disk *tag*. The source disk is expected to be accessible to rsync as *srchost*::*tag*, so there should be an rsync server daemon running on the source machine, offering the source disk as rsync module *tag*. This can be set up using the **rsync** LCFG component.

dstdir_*tagtag!rmirror resource*

The local destination directory holding the mirror for disk *tag*.

checksum

When set to *true* rmirror tells **rsync** to use the **–checksum** option. When rsync is deciding which files need to be transferred to bring the mirror up to date, this makes rsync checksum all files before transfer, and transfer any files whose checksum and/or size does not match that of the corresponding file already on the rmirror. This option effectively makes rmirror more meticulous when checking for file corruption.

When set to *false* checksum-checking will not be used on files whose mirror has the same timestamp and size.

The default value is *false*.

checksum_*tagtag!rmirror resource*

When set to *true* this option overrides the current setting of the **checksum** resource for disk *tag* only. Its default value is the same as that of the **checksum** resource.

timestamp

When set to *true*, **rsync** will not copy any files whose existing mirror copies are already the same length and have the same timestamp.

When set to *false* rmirror tells **rsync** to use the **–ignore-times** option, which makes rsync ignore timestamps when deciding which files to copy.

The default value is *true*.

timestamp_*tagtag!rmirror resource*

When set to *true* this option overrides the current setting of the **timestamp** resource for disk *tag* only. Its default value is the same as that of the **timestamp** resource.

wholefiles

When set to *true* this option tells **rsync** to use its **–whole-file** option. This makes **rsync** copy across whole files which have changed, rather than using its incremental algorithm to copy across only the changes.

When set to *false* **rsync** will use its default behaviour, which is to use its incremental algorithm - that is, copying across only the parts of files which have changed - unless both source and target are on the local machine. This option will often result in quicker running of rmirror, but may lead to any temporary corruption of a file on the source machine persisting indefinitely in the target machine's copy of the file.

The default value is *false*.

wholefiles_*tagtag!rmirror resource*

When set to *true* this option overrides the current setting of the **wholefiles** resource for disk *tag* only. Its default value is the same as that of the **wholefiles** resource.

deleteafter

When set to *true* this option makes **rmirror** tell **rsync** to use its **–delete-after** option. This makes **rsync** delete outdated files on the target system after copying across changes from the source system.

When set to *false* this option makes **rmirror** tell **rsync** to use its default file deletion behaviour, which is to delete outdated files on the target system before copying across changes from the source system.

The default value is *false*.

deleteafter_*tagtag!rmirror resource*

When set to *true* this option overrides the current setting of the **deleteafter** resource for disk *tag* only. Its default value is the same as that of the **deleteafter** resource.

timeout

This resource is the time, in seconds, before rmirror gives up on an apparently dormant rsync process on a remote machine, times it out, and goes on to the next backup to be performed. The **timeout** option sets the timeout value for all rmirror backups on this machine. The default value is *3600*, meaning one hour. To disable timeouts set this option to 0.

timeout_*tagtag!rmirror resource*

This resource overrides the current setting of the **timeout** resource for disk *tag* only. It has the same default value as the **timeout** resource.

safetylimit

This resource is the maximum percentage of the files in the existing backup copy that it is permissible for an rmirror run to delete. Just before rmirror performs a backup, it calculates what percentage of the files in the existing backup would be deleted by the backup running again; and if this percentage is greater than the maximum allowable percentage in the **safetylimit** resource, the backup is cancelled and a warning is given. The default value of **safetylimit** is 10, meaning that a backup will not run if it would mean deleting more than 10% of the existing backup files.

safetylimit_*tagtag!rmirror resource*

This resource overrides the current setting of the **safetylimit** resource for disk *tag* only. It has the same default value as the **safetylimit** resource.

AUTHORS

```
Chris Cooke <cc@inf.ed.ac.uk>
```

VERSION

1.8.9-1

B.48 routing

The LCFG routing component

DESCRIPTION

This object constructs all the necessary configuration files and starts the appropriate routing daemon.

GENERIC RESOURCES

type

Do we want to run `routed` or `gated` or (eventually) `zebra`?

If we run routed then we just have to accept everything that's thrown at us. If we run gated then we do a bit more work, but we also get more control over what we accept.

If this is null then the object gets to choose what's "best" (but note that it will then force rdisc off).

contextlabel

This resource does not actually affect the operation of the component, but instead is included in some of its messages. Setting it to some lcfg context-specific value might therefore be useful to the user.

snmp

Do we want the daemon to attempt to speak snmp? (Probably not!) This is in principle a generic resource, though only gated understands it at the moment.

ROUTED RESOURCES

routed_binary

The name of the `routed` binary.

GATED RESOURCES

static

Specifies hosts and networks for which static routes should be installed. `static` contains a list of tags. For each *tag* there must be a corresponding `gateway_`*tag* and optional `hosts_`*tag* and/or `networks_`*tag*. The former is just a list of host IP addresses; the latter is a list of networks, with optional masks separated by ':' or mask lengths separated by '/'.

rip_import

rip_import_extra

List of networks which should be imported from RIP (with optional masks or lengths separated by ':' or '/' as before). If this is blank then we just accept everything we're given. `rip_import_extra` has two functions: it makes it easy to add things to the default set, and it means these resources can each be short enough to fit even though the combined length is too much.

rip_accept_default

Should we accept the default route? Set this to null to ignore it.

rip_nobroadcast

Should we send rip packets? If this is set then we don't.

rip_ifs

Since we can only have one "interface" statement for each interface we bundle the functionality under the `rip_ifs` resource, which contains a list of interfaces for which metric-tweaking is required. For each there's then a corresponding `rip_metricin_`*if* and `rip_metricout_`*if* resource, which control the metric which should be set on input or added on output, and `rip_ripin_`*if* or `rip_noripin_`*if*, which control whether RIP is accepted or not and which can't both be set. Likewise `rip_ripout_`*if* and `rip_noripout_`*if* control the sending of routing information. At least one of the interface sub-resources has to be set. The usual gated rules apply here; in particular "all" is acceptable. The tag-name is assumed to be the interface name by default, but if this isn't appropriate then the `rip_ifname_`*if* resource can be used to change it. Finally, we may want to assign a non-default metric using `if_metric_`*if* when we export it as a direct route.

rip_export

rip_export_extra

rip_exportifs

There are two lots of resources involved in RIP exporting: `rip_export` and `rip_export_extra`, if set, define the defaults for all not-otherwise-specified interfaces; and `rip_exportifs` contains a list of interfaces for specific handling, each of which has a corresponding `rip_export_`*whatever* list. If it's required to control explicitly which directly-connected networks should be exported everywhere, this can be done by setting the appropriate `rip_export_direct_`*whatever* resources. `rip_descr_`*whatever* adds a helpful comment to the gated configuration file. `rip_name_`*whatever* sets the interface name, if it's different from the tag.

Note that it may be necessary for *whatever* to be "all".

rdisc_server

Should we run rdisc? Note that this'll be unconditionally forced off unless `type` is explicitly set to `gated`.

gated_binary

The name of the `gated` binary.

gated_pid_file

Where `gated` will write its pid file.

gated_config_file

Where should the generated `gated.conf` file go?

gated_syslog_level

At what syslog level should `gated`'s messages be produced?

traceoptions

rip_traceoptions

Trace options, in `gated`-standard form. `traceoptions` specifies global options, while `rip_traceoptions` specifies RIP-specific options.

STATIC RESOURCES

static_default

The address of a router which should be set as the static default.

PRIVATE RESOURCES

The following resources should not normally have their values changed from the installation defaults. They are use either to communicate state between method invocations, or to define where the component's various compiled C helper programs have been installed, or to provide Solaris/Linux compatibility hooks. Setting them incorrectly may result in the component not functioning correctly. Refer to the component source itself for details as to their various functions.

checkInList

AUTHORS

```
George Ross &lt;gdmr@dcs.ed.ac.uk&gt;
```

VERSION

3.3.40-1

B.49 rpmaccel

LCFG rpmaccel component

DESCRIPTION

This component configures a squid accelerator for fronting an RPM repository.

httpport

The port that the squid accelerator should listen on. Defaults to 80.

cachemem

Specifies a limit on how much additional memory squid uses as a memory cache of objects. Sets the squid `cache_mem` parameter. Defaults to 128 Mb.

maxobjsize

Objects larger than this size will not be cached. Sets the squid `maximum_object_size` parameter. Defaults to 1024 Mb.

cachedir

The location of the cache directory tree. Used to set the squid `cache_dir` parameter. Defaults to `/var/spool/squid`.

cachedirsize

Specifies the maximum amount of disk space the cache should occupy in the cache directory tree. Defaults to 10 Gb.

accelhost

Specifies the hostname of the HTTP server that is to be *accelerated*. Sets the squid `httpd_accel_host` parameter. No default value.

accelport

Specifies the port number of the HTTP server that is to be *accelerated*. Sets the squid `httpd_accel_port` parameter. Defaults to 80.

acltags

A list of squid ACLs.

acltag_*aclname*

The ACL value for the ACL with name *aclname*.

accesstags

A list of squid http_access rules.

accesstag_*tag*

The value of the squid http_access rule with tag *tag*.

AUTHORS

```
Alastair Scobie &lt;ascobie@inf.ed.ac.uk&gt;
```

VERSION

0.99.3-1

B.50 rpmcache

LCFG component to maintain a local RPM cache

DESCRIPTION

This component maintains a cache directory on the local machine containing a copy of every RPM specified in an rpmcfg file. The list of available RPMs is obtained by reading a file called **rpmlist** from the server in exactly the same way as **updaterpms**. Wildcards in the rpmcfg file are evaluated, and the cache is updated by fetching RPMs from the repository over HTTP. Normally, the rpmcfg file will be the client's own rpmcfg file so that the cache contains a copy of every RPM that should be installed on the system. The **updaterpms** component may then use the cache as the source directory, allowing the system to be updated without direct access to a remote repository.

The **run** method is used to initiate a cache update and supports the following options:

-c

Force old entries to be removed (cleaned) from the cache, even if the **preserve** resource is set.

-p

Do not delete (preserve) old entries in the cache, even if the **preserve** resource is not set.

-r *root*

root is prefixed to the **cachedir** resource to form the name of the cache directory. This is useful at install time.

-t

Test only. Display cache operations that would be performed without performing them.

The **install** method is identical to the **run** method, except that the resources are loaded directly from the profile (rather than the status file), and no locking or status saving takes place. This is intended for use at install time (only).

RESOURCES

cachedir

The pathname of the cache directory on the client.

cppopts

Additional options to be passed to cpp when reading rpmcfg files.

genhdfile

If this resource is set, then the **genhdfile** program is run on rpms after downloading them, to create the header info file.

localpath

A (space-separated) list of directories on the client machine to be treated as local master repositories. Any RPMs present in these directories will not be searched for on the server, and will not be copied into the cache.

preserve

Do not delete old entries from the cache.

rpmlist

The name of a file to be created in the cache directory, containing a list of the RPMs in the cache. By default, this is null and no file is created.

rpmpath

A comma separated list of URLs for directories containing RPMs. Each directory should contain a set of RPMs, and a file named **rpmlist**, listing the available RPMs. Local directory names may also be specified instead of URLs.

rpmcfg

The full pathname of the rmcfg file on the client machine containing the list of RPMs to maintain in the cache. Multiple (comma-separated) rpmcfg files may be specified, in which case they are effectively concatenated. rpmcfg files may also be specified as remote URLs which are automatically downloaded - however, note that any files included with **#include** will not be automatically downloaded.

rpmlock

The name of a lock file in the cache. This file will be removed before a cache update, and created after the update. This prevents **updaterpms** from atempting to run during a cache update.

trigger

A command run run after a sucessful update of the cache. For example (and by default) **om updaterpms run**.

PLATFORMS

Redhat7, Redhat9, Solaris9

AUTHOR

Paul Anderson <dcspaul@inf.ed.ac.uk >

VERSION

1.1.17-1

B.51 rsync

LCFG rsync server daemon component

DESCRIPTION

This component configures and starts the rsync server daemon. Changes to LCFG resources will automatically result in changes to the rsync server's configuration.

RESOURCES

globals

A list of tags, one for each *gentry* resource.

gentry_*tag*

An rsync global configuration option. For more details of rsync's possible global configuration options see the man page for *rsyncd.conf*. *tag* should be listed in the *globals* resource.

modules

A tag list of the rsync modules to be set up; one tag for each rsync module.

mentries_*module*

For rsync module *module*, a tag list of the configuration options for that module. *module* should be listed in the *modules* resource.

mentry_*module*_*tag*

A configuration option for rsync module *module*. *tag* should be listed in resource mentries_*module*. For details of rsync's possible global configuration options see the man page for *rsyncd.conf*.

EXAMPLE

rsync.globals log rsync.gentry_log log file = /var/obj/log/rsync

rsync.modules glasgow rsync.mentries_glasgow 1 2 3 4 5 rsync.mentry_glasgow_1 path=/disk/home/glasgow rsync.mentry_glasgow_2 hosts allow=foo.inf.ed.ac.uk bar.inf.ed.ac.uk rsync.mentry_glasgow_3 hosts deny=* rsync.mentry_glasgow_4 uid=0 rsync.mentry_glasgow_5 read only=yes

AUTHORS

```
Chris Cooke <cc@inf.ed.ac.uk>
```

VERSION

2.1.0-1

B.52 server

LCFG server component

DESCRIPTION

The profile server component for LCFG. This component manages the mxkprof daemon which compiles configuration protocols. Note that the **server** resources control the actions of the server and are only required on hosts running a **server** component. However, the server does require some resources for each client that it compiles; these resources are the **profile** resources.

ADDITIONAL METHODS

The **run** method sends a HUP to the mkxprof daemon to initiate a recompilation. Option **-r** sends a HUP to the daemon after creating a flag file which causes it to completely rebuild all dependencies and profiles.

RESOURCES

acl_*access_fileaccess_file!server resource*

An Apache "**allow from**" specification for the access file corresponding to *access_file*, specifying hosts which should be permitted to access this directory without authentication.

auth_*dirdir!server resource*

A list of tags representing Apache web access files to be created in the directory corresponding to *tag* (see **linkdirs**).

debug

A set of `mkxprof` debug flags.

defpath

A space separated list of directory pathnames in which to search for component default files. The directories are searched in the given order. The default is /usr/lib/lcfg/defaults/server. Default files must have the extension **.def**.

derive

If this resource is non-null, mkxprof will generate derivation attributes in the profile (**-r** option).

dst_*tagtag!server resource*

The destination directory to be created for the given *tag* (see **linkdirs**).

fetch

A space separated list of specifications of the form **dst**=**src**, where **src** and **dst** are rsync specifications. Rsync will be called for each item in this list before starting a compilation cycle. This allows source directories to be collated from multiple remote servers.

file_*access_fileaccess_file!server resource*

The name of the Apache access file for the given *access_file*. Normally, this will be **.htaccess**, but Apache may be configured to use multiple different access files in different circumstances: for example, a different access file may be used for SSL and plain HTTP.

hdrpath

A space separated list of directory pathnames in which to search for header files. The directories are searched in the given order. The default is /var/lcfg/conf/server/include,/usr/lib/lcfg/server/include. Header files must have the extension **.h**.

linkdirs

A (space-separated) list of tags representing directories to be created in the root web directory. Each directory is automatically populated with links to the contents of some other directory, and one or more Apache access control files may be automatically created. By default, this is used to export the icons used by the status pages, and the CGI scripts. Other tags can be added to export, for example, an RPM repository, or some other static pages.

lockfiles

The **lockfiles** resource is comma-separated list of full pathnames for lock files. In daemon mode, **mkxprof** will not compile source files while any of these lockfiles exist; the compilation will be deferred until the next poll, or notification. This provides a mechanism to allow several synchronized changes to be made to related files, in an atomic way.

poll

The poll (**-p**) argument for mkxprof.

pkgpath

A space separated list of directory pathnames in which to search for package lists. The directories are searched in the given order. The default is /var/lcfg/conf/server/packages. Package lists must have the extension **.pkgs** or **.rpms**.

pwf_*access_fileaccess_file!server resource*

The name of an Apache DB password file which will be used to to authenticate requests for files in this directory when the access control conditions are not satisified (see **acl**_*access_file*), or not present. Any valid user in the password file will be permitted to connect. The special value **auto** can be supplied in which case the server will use the password file created automatically from the **profile.passwd** resources.

ropts

Additional rsync options for mkxprof (**-o** option).

servername

The FQDN of the server to be used in status messages and profiles. The default is obtained from the **hostname** command.

src_*tagtag!server resource*

The source directory to be created for the given *tag* (see **linksdirs**). All items in this directory will be symbolically linked to corresponding items in the destination directory.

srcpath

A space separated list of directory pathnames in which to search for source files. The directories are searched in the given order. The default is /var/lcfg/conf/server/source. Source files must have no extension.

statichtml

This option generates static HTML pages containing status information (**-s** option). Normally, the **status** resource should be used instead .

status

If this option is present, status information, including all errors and warnings are stored for display on an HTML status page. Normally, the CGI scripts **status** and **index** will be used to display this status information. The **statichtml** resource can be used to automatically generate static HTML pages which do not require the CGI sripts, however this is less flexible and results in slower compilations (**-h** option).

stats

If this resource is non-null, mkxprof will write statistics to the logfile /var/lcfg/log/server.stats (**-x** option).

valpath

A space separated list of directory pathnames in which to search for validation files. The directories are searched in the given order. The default is /var/lcfg/conf/server/validation. These files are used by the **vINFILE** macro for validating strings.

verbose

Non-null for mkxprof verbose logging.

warn

mkxprof warning flags.

webdir

The web directory for the **-w** option of mkxprof. The default is /var/lcfg/conf/server/web.

SEE ALSO

lcfg-profile

The definition of client resources used by the server.

PLATFORMS

Redhat7, Redhat9, Solaris9

AUTHOR

Paul Anderson <dcspaul@inf.ed.ac.uk >

VERSION

2.1.64-1

B.53 snmp

The LCFG SNMP component

DESCRIPTION

This object constructs all the necessary configuration files and starts the `snmp` local agent and optionally the trap daemon.

RESOURCES

daemon

ucdv4snmpd

Which type of daemon should we run? The component itself may support more than one, depending on the platform. And where do we find it?

killsig

When stopping, what signal would the daemon like?

read_community

trap_community

Read and trap communities. We don't enable write!

send_traps

trapHosts

Should we send traps? If so, where to? (This is a good candidate for being context-sensitive.)

local_net

For access control, a list of local networks.

sysDesc

sysLocation

make

model

sno

hostid

sysContact

Various useful things to put in the MIB variables.

AUTHORS

```
George Ross <gdmr@dcs.ed.ac.uk>
```

VERSION

3.1.4-1

B.54 sshd

LCFG SSHD Component

SYNOPSIS

sshd *METHOD* [*ARGS*]

DESCRIPTION

An LCFG component that is used to configure and manage the SSH daemon and the generation/publication of keys.

METHODS

The non-standard component methods are described below.

GenerateKeys

> Generates an RSA key and a DSA key and publishes the keys into a central register using the program defined in the *keydisthelper* resource.

RemoveKeys

> Removes any existing RSA key and DSA key and deletes the keys from a central register using the program defined in the *keydisthelper* resource.

RESOURCES

The non-standard component resources are described below.

keydisthelper

> The full path to a script which manages the distribution of SSH keys. If empty (the default), no distribution is performed.
>
> The script must take a number of command line options:
>
> –add1 *file*
>
>> Add the version 1 key contained in *file* to the distribution for this machine.
>
> –add2 *file*
>
>> Add the version 2 key contained in *file* to the distribution for this machine. Note that –add2 may be specified multiple times on machines which have multiple keys.
>
> –extract1 *file*
>
>> Create a version 1 known_hosts file in *file*.
>
> –extract2 *file*
>
>> Create a version 2 known_hosts file in *file*.
>
> –delete
>
>> Delete all SSH keys for this machine.

krb5auth

> This sets the sshd_config `Krb5Authentication` variable. Setting this to `yes` will allow kerberos authentication.

krb5tgtpass

This sets the sshd_config `Krb5TgtPassing` variable. Setting this to `yes` will allow KerberosTGT forwarding to the server.

rhostsrsa

This sets the sshd_config `RhostsRSAAuthentication` variable. Leaving this empty or set to `yes` will allow rhosts or /etc/hosts.equiv authentication together with successful RSA host authentication. Setting this to `no` will disallow this.

FILES

/etc/ssh/ssh_host_key

/etc/ssh/ssh_host_dsa_key

/etc/ssh/ssh_known_hosts

/etc/ssh/ssh_known_hosts2

/etc/ssh/ssh_config

/etc/ssh/sshd_config

/usr/lib/lcfg/conf/sshd/ssh_keys_published

PLATFORMS

Redhat9

SEE ALSO

sshd, ssh-keygen, kinit, ldapsshkeys

AUTHOR

DICE Authentication and Authorization Team <auth-team@inf.ed.ac.uk >

VERSION

1.20.4-1

B.55 symlink

LCFG symlink component

DESCRIPTION

This component builds and removes symbol links.

links

A list of symbol links to make. Each link has a linkname_*link* tag and a target_*link* tag.

linkname_*link*

The linkname of the link specified by tag *link*.

target_*link*

The target of the link specified by tag *link*.

zap_*link*

Normally the symlink component will fail if there is already a real file or directory with the same name as the `linkname`. Setting this resource to `yes`, overrides this behaviour and instructs the component to zap any preexisting file or directory. Dangerous !

AUTHORS

`Alastair Scobie <ascobie@inf.ed.ac.uk>`

VERSION

0.100.6-1

B.56 syslog

LCFG syslog component

DESCRIPTION

This component configures and starts the `syslogd` daemon and starts the `klogd` daemon. It is normally the first component to be run (as specified in the resource `boot.services`).

The base `syslog.conf` configuration file is created from a template that allows for almost complete control of the contents of the configuration file. The contents are controlled via resources described below. The base file generated from the template is passed through `m4` when `syslog` is configured or started. A number of symbols are predefined, but these do not include the `LOGHOST` variable which is conventionally available.

New rules are added to the syslog.conf file by defining the selector and action fields for the rule (see syslog.conf(5)) and usually also the text of a preceding comment line. The rules are grouped in four sections corresponding to the resources `priorities`, `applications`, `otherlines` and `additions`.

RESOURCES

Syslogd Configuration File Resources

priorities

A list of *tags* for rules concerning messages of different priority levels to be included in `syslog.conf`. Default value is the list `emerg alert err`.

pricomment_*tag*

The text for a comment line that precedes the rule for priority *tag*.

pricomment_emerg defaults to `Emergency messages will be displayed using wall`.

pricomment_alert defaults to <Alert messages will be directed to the operator >.

pricomment_err defaults to `Errors go to the console`.

priselector_*tag*

The *selector* field of the rule for priority *tag*.

priselector_emerg defaults to `*.emerg`.

priselector_alert defaults to `*.alert`.

priselector_err defaults to `*.err`.

priaction_*tag*

The *action* field of the rule for priority *tag*.

priaction_emerg defaults to `*`.

priaction_alert defaults to `root`.

priaction_err defaults to `/dev/console`.

applications

A list of *tags* for rules concerning messages from different *facilities* to be included in `syslog.conf`. Default value is the list `auth authpriv mail local1 local2 local6`.

appselector_*tag*

The *selector* field of the rule for facility *tag*.

appselector_auth defaults to `ifdef('AUTHDEBUG',auth.debug,auth.info)`.

appselector_authpriv defaults to `ifdef('AUTHDEBUG',authpriv.debug,authpriv.info)`.

appselector_mail defaults to `ifdef('MAILDEBUG',mail.debug,mail.info)`.

appselector_local1 defaults to `ifdef('INETDEBUG',local1.debug,local1.info)`.

appselector_local2 defaults to `ifdef('XNTPDDEBUG',local2.debug,local2.notice)`.

appselector_local6 defaults to `ifdef('DNSDEBUG',local6.debug,local6.info)`.

appaction_*tag*

The *action* field of the rule for facility *tag*.

appaction_auth defaults to `AUTHLOG()`.

appaction_authpriv defaults to `AUTHLOG()`.

appaction_mail defaults to `MAILLOG()`.

appaction_local1 defaults to `INETLOG()`.

appaction_local2 defaults to `XNTPDLOG()`.

appaction_local6 defaults to `DNSLOG()`.

otherlines

A list of *tags* for miscellaneous other rules. Defaults to `local7 general`.

othcomment_*tag*

The text for a comment line that precedes the rule corresponding to *tag*.

othcomment_local7 defaults to `Messages from init are reported on local7 (as defined by /etc/initlog.conf)`.

othcomment_general defaults to `Other general messages go to the syslog log file`.

othselector_*tag*

The *selector* field of the rule corresponding to *tag*.

othselector_local7 defaults to `local7.info`.

othselector_general defaults to `*.info;kern.info;auth.none;mail.none;local1.none;local5.none;local6.none`

othaction_*tag*

The *action* field of the rule corresponding to *tag*.

othaction_local7 defaults to `BOOTLOG()`.

othaction_general defaults to `LOGFILE()`.

additions

A list of *tags* for additional rules to be added to `syslog.conf`.

addcomment_*tag*

The text for a comment line that precedes the additional rule corresponding to *tag* to be added to the `syslog.conf` file.

addselector_*tag*

The *selector* field of the additional rule corresponding to *tag* to be added to the `syslog.conf` file.

addaction_*tag*

The *action* field of the additional rule corresponding to *tag* to be added to the `syslog.conf` file.

add_*tag*

The complete additional rule corresponding to *tag* to be added to the `syslog.conf` file. This is for backwards compatability with earlier versions of the syslog component.

Other Resources

sopts

Options for the `syslogd` daemon.

kopts

Options for the `klogd` daemon.

m4_defines

Additional definition macros for m4 when generating the `syslog.conf` file from the base configuration file derived from the template.

This can be used to change the priority level of messages logged for the facilities controlled via the `application` resource. For example using the default value for appselector_mail one could change the priority level of messages being logged by setting this resource to `-DMAILDEBUG`.

PLATFORMS

Redhat7, Redhat9

AUTHOR

Alastair Scobie <ascobie@inf.ed.ac.uk >, Ken Dawson <ktd@inf.ed.ac.uk >

VERSION

1.1.0-1

B.57 tcpwrappers

LCFG tcpwrappers component

DESCRIPTION

This component configures the machine's tcpwrappers.

allow

A list of services that are to be included in the `/etc/hosts.allow` file for access control by `tcpd`.

allow_*service*

The access control list for the specified service. This is a list of patterns as specified in the man page for `hosts_access`.

deny

A list of services that are to be included in the `/etc/hosts.deny` file for access control by `tcpd`.

deny_*service*

The access control list for the specified service. This is a list of patterns as specified in the man page for `hosts_access`.

banners

A list of services that will have `/etc/tcp.banners/` entries created.

banlines_*service*

A list of banner lines tags for service `service`

banline_*service_tag*

The banner line associated with tag *tag*.

AUTHORS

```
Alastair Scobie <ascobie@inf.ed.ac.uk>
```

VERSION

0.99.5-1

B.58 toshset

The LCFG toshset component

DESCRIPTION

This component controls the **toshset** utility to change the parameters of a toshiba laptop depending on the power state. The **configure** method should be caled from the apm component when the power state changes.

RESOURCES

battery

A list of tags for parameters to be passed to toshset when running on battery.

battery_*tagtag!toshset resource*

The arguments to be passed to toshset for the specified tag.

context_battery

An LCFG context to be enabled when running on battery.

context_line

An LCFG context to be enabled when running on mains power.

line

A list of tags for parameters to be passed to toshset when running on mains power.

line_*tagtag!toshset resource*

The arguments to be passed to toshset for the specified tag.

PLATFORMS

Redhat9

AUTHOR

Paul Anderson <dcspaul@inf.ed.ac.uk >

VERSION

0.99.3-1

B.59 updaterpms

LCFG updaterpms component

DESCRIPTION

This object is used to manage the installed RPMs.

The `start` and `run` methods attempt to match the installed RPMs with those listed in the machine's RPM spec file.

The `install` method is used to install a Linux box. It only works if the running system filesystems are not the same as the destination system filesystems. It is usually run only from the install subsystem, with network system filesystems.

The `testrpm` method can be used to check which RPMs will be installed, upgraded or removed when the `start` or `run` methods are invoked.

The `deleterpm` method can be used to delete a specified RPM manually. This is useful when the updaterpms process has gone AWOL for some reason and one needs to fix up a lot of machines (via `om`). Very rarely used.

The `installrpm` method can be used to install a specified RPM manually. This is useful when the update process has gone AWOL for some reason and one needs to fix up a lot of machines (via `om`).

offline

This resource, if set to `yes`, stops the object from making any configuration changes when the `start` method is invoked. This is handy for portables where the `run` method is user invoked to make configuration changes.

cppbin

The pathname of the preprocessor used to preprocess the RPM spec file.

flags

Extra flags to be used for `updaterpms`.

rpmdir

The directory containing the RPMs.

rpmcfgdir

The directory containing the RPM spec files.

rpmcfg

The RPM spec file for this machine.

xferdir

Temporary directory for downloading RPMs fetched over HTTP Defaults to */var/tmp*

rpmlock

This resource, if defined, specifies a lock file to look for on the RPM repository before running updaterpms. If the file is missing, it is assumed that an RPM repository update is in progress and updaterpms won't be run.

mail

Specifies who should be mailed if the software update process fails.

AUTHORS

```
Alastair Scobie <ajs@dcs.ed.ac.uk>
```

VERSION

0.100.29-1

B.60 vigor

Configure Vigor 2600 router

DESCRIPTION

This component is intended to monitor and configures a Vigor 2600 DSL router. Router configuration is not yet implemented and the component currently only processes syslog messages from the router.

The 2600 router sends a large number of syslog messages to various hardwired facilities and priority levels. If these are processed by the normal syslog daemon, they tend to swamp important messages generated by other services, and they are difficult to filter. The **lcfg-vigor** component acts as a syslog daemon on an alternative port which can be used to process just vigor messages. Error and Warning messages generate LCFG errors and warnings which appear in the status display. The WAN pings are also monitored to flag the router or link as down.

RESOURCES

ackinterval

The maximum delay between WAN pings and corresponding ACKs before flagging the link as down.

inbound

True to log inbound conections.

ip

The IP address of the router.

outbound

True to log outbound conections.

pinginterval

How long (in seconds) between WAN pings should be allowed before flagging the router as down.

pollinterval

How often (in seconds) to check the WAN ping responses.

port

The port for the syslog daemon. This should match the value configured into the router. The default is 735.

waninfo

True to log chatty info messages about the WAN (very frequent pings!).

PLATFORMS

Redhat7, Redhat9

AUTHOR

Paul Anderson <dcspaul@inf.ed.ac.uk >

VERSION

0.99.12-1

B.61 vlan

LCFG vlan component

DESCRIPTION

This component configures VLAN interfaces.

The current version is intended to be used at system boot only.

vlans

A list of VLANs to be configured. Each VLAN must have the following resources.

interface_*vlan*

The physical interface to be configured with this VLAN.

tag_*vlan*

The VLAN numerical tag for this VLAN.

nametype

Sets the way vlan-device names are created. See the man page for `vconfig`.

AUTHORS

```
Alastair Scobie <ascobie@inf.ed.ac.uk>
```

VERSION

0.100.0-1

B.62 vmidi

Configure external MIDI device

DESCRIPTION

This component uses the v_midi kernel module to drive an external (serial or parallel) MIDI controller. The kernel module is loaded and a daemon copies data from the vmidi output to the specified port.

RESOURCES

dev

The device to which the MIDI controller is attached (eg. ttyS0).

rate

The baud rate for the device (eg. 38400). This must be null for non-serial devices.

PLATFORMS

Redhat7, Redhat9

AUTHOR

Paul Anderson <dcspaul@inf.ed.ac.uk>

VERSION

0.99.6-1

B.63 xfree

LCFG xfree component

DESCRIPTION

This component builds the XFree86 `/etc/X11/XF86Config` file. It does not start the X server.

This text documents the resources used to construct the `/etc/X11/XF86Config` file. It does not explain the semantics of the `/etc/X11/XF86Config`; for this you should read the **XF86Config** man page.

RESOURCES

Resources are grouped by the **XF86Config** section they construct.

Section "Files"

fontpaths

A list of fontpaths.

fontpath_*fp*

The **FontPath** entry for fontpath *fp*.

modulepaths

A list of modulepaths.

modulepath_*mp*

The **ModulePath** entry for modulepath *mp*.

rgbpath

The **RGBPath** entry.

Section "ServerFlags"

flags

A list of server flags.

flag_*f*

The **Option** flag name for flag *f*.

flagvalue_*f*

The **Option** flag value for flag *f*.

Section "Module"

modules

A list of module. Used to generate **Load** and **SubSection "module"** entries.

modopts_*m*

A list of module option tags for module *m*. Used to generate the **Option** entries within a module's **SubSection** entry.

modopt_*m*_*mo*

The **Option** entry associated with the module *m* and tag *mo*.

Section ”InputDevice”

inputdevices

A list of inputdevice tags. Used to generate one or more **Section ”InputDevice”** blocks.

Each tag *id* is used to generate the **Identifier** entry in the relevant block, generating the name **input**_*id*.

inputdriver_*id*

The **Driver** entry for the input device *id*.

inputopts_*id*

A list of inputdevice options for the input device *id*.

inputopt_*id*_*io*

The options entry associated with the input device *id* and tag *io*. Note that the resource value is inserted verbatim.

Section ”Device”

devices

A list of video device tags. Used to generate one or more **Section ”Device”** blocks.

Each tag *vd* is used to generate the **Identifier** entry in the relevant block, generating the name **video**_*vd*.

videodriver_*vd*

The **Driver** entry for video device *vd*.

videoscreen_*vd*

The **Screen** entry for video device *vd*.

vidopts_*vd*

A list of video options for video device *vd*.

vidopt_*vd*_*do*

The options entry associated with video device *vd* and tag *do*. Note that the resource value is inserted verbatim.

Section ”Monitor”

monitors

A list of monitor tags. Used to generate one or more **Section ”Monitor”** blocks.

Each tag *mid* is used to generate the **Identifier** entry in the relevant block, generating the name **monitor**_*mid*.

hsync_*mid*

The **Horizsync** entry for monitor *mid*.

vrefresh_*mid*

The **Vertrefresh** entry for monitor *mid*.

monopts_*mid*

A list of options for monitor *mid*.

monopt_*mid*_*mo*

The options entry associated with monitor *mid* and tag *mo*. Note that the resource value is inserted verbatim.

Section "Modes"

modegrps

A list of mode groups. Used to generate one or more **Section "Modes"** blocks.

Each tag *mg* is used to generate the **Identifier** entry in the relevant block, generating the name **modegrp**_*mg*.

modes_*mg*

A list of modelines for the mode group *mg*.

mode_*mg*_*mm*

The **Modeline** entry associated with mode group *mg* and tag *mm*.

Section "Screen"

screens

A list of screens. Used to generate one or more **Section "Screen"** blocks.

Each tag *sid* is used to generate the **Identifier** entry in the relevant block, generating the name **screen**_*sid*.

device_*sid*

The video device to use for screen *sid*. This must refer to an entry from the resource **xfree.devices**.

monitor_*sid*

The monitor to use for screen *sid*. This must refer to an entry from the resource **xfree.monitors**.

displaydepth_*sid*

The **DefaultDepth** resource for screen *sid*.

screenopts_*sid*

A list of screen options for screen *sid*.

screenopt_*sid*_*so*

The options entry associated with screen *sid* and tag *so*. Note that resource value is inserted verbatim.

displaydepth_*sid*

The **Depth** value for the **SubSection "Display"** block for screen *sid*.

displaymodes_*sid*

The **Modes** value for the **SubSection "Display"** block for screen *sid*.

displayopts_*sid*

A list of display options for the **SubSection "Display"** block for screen *sid*.

displayopts_*sid*_*do*

The display options entry associated with screen *sid* and tag *do*. Note that resource value is inserted verbatim.

Section "ServerLayout"

layouts

A list of layouts. Used to generate one or more **Section "ServerLayout"** blocks.

Each tag *lid* is used to generate the **Identifier** entry in the relevant block, generating the name **layout_***lid*.

layoutscreens_*lid*

A list of **Screen** entries for layout *lid*. Each entry must refer to a screen defined in the resource **xfree.screens**. The entry name is used to generate the *screen-id* field of the **Screen** entry.

layoutscreenid_*lid*_*sid*

The *screen-num* for the **Screen** entry associated with layout *lid* and entry *sid*.

layoutscreenpos_*lid*_*sid*

The position information for the **Screen** entry associated with layout *lid* and entry *sid*.

layoutinputs_*lid*

A list of **InputDevice** entries for layout *lid*. Each entry must refer to an inputdevice defined in the resource **xfree.inputdevices**. The entry name is used to generate the *idev-id* field of the **InputDevice** entry.

layoutinputopts_*lid*_*iid*

Additional option fields for the **InputDevice** entry associated with layout *lid* and tag *iid*.

layoutopts_*lid*

A list of layout options for layout *lid*.

layoutopt_*lid*_*lo*

The options entry associated with layout *lid* and tag *lo*. Note that resource value is inserted verbatim.

Section "DRI"

drimode

The **DRIMode** entry of **Section "DRI"**.

drigroup

The **DRIgroup** entry of **Section "DRI"**.

AUTODETECTION

If the resource **xfree.device_main** has the value **auto**, the component will attempt to identify the driver for the system's video card and set the resource **xfree.videodriver_auto** appropriately. This assumes that there is a screen called *main* and a video device called *auto*.

If the resource **xfree.monitor_main** has the value **auto**, the component will attempt to identify the system's monitor will set the resources **xfree.hsync_auto** and **xfree.vrefresh_auto** appropriately. This assumes that there is a screen called *main* and a monitor called *auto*.

FILES

/etc/X11/XF86Config

The XF86Config file generated by this component.

/usr/lib/lcfg/conf/xfree/templates/xfree.conf.tmpl

The LCFG sxprof template which processes the above resources.

PLATFORMS

Redhat9

AUTHOR

Alastair Scobie <ascobie@inf.ed.ac.uk >

VERSION

1.0.0-1

B.64 xinetd

The lcfg xinetd component. xinetd is the extended Internet services daemon.

DESCRIPTION

This component starts, stops and configures the xinetd daemon. The configuration file used by xinetd is specified by the *conffile* resource (default is `/etc/xinetd.conf`). Services generally provide their own configuration file, located in the directory specified by the *basedir* resource (default is `/etc/xinetd.d/`). These values can be overridden or specified in their entirety by resources, to produce a final configuration file located in the directory specified by the *confdir* resource (default is `/etc/xinetd.lcfg/`).

RESOURCES

conffile

The configuration file used by the xinetd daemon. Typically this file contains some default values and then specifically includes the directory containing individual service definitions. Default is `/etc/xinetd.conf`.

basedir

The base directory where vendor-provided service definition files can be found. The default is `/etc/xinetd.d/`. It is very unlikely that you will want to change this value.

confdir

The configuration directory that will be used by the xinetd daemon for service definition files - it will be included at the end of the main configuration file (see *conffile* above). The definitions in this directory can be a combination of files sourced from the directory specified by *basedir*, those specified entirely by resources (see below), or a combination of the two. Resources defined for a particular service will override the default configuration for that service at the attribute level. The component is responsible for producing the files in this directory.

enableservices

A list of services to be enabled. Only items in this list will be enabled, regardless of any other factors. **Important note:** For services that provide their own definition file (in the directory specified by *basedir*, typically `/etc/xinetd.d/`), the *name of the file* should be used in *enableservices* and also in *services* (should any of its attributes require to be overridden). This is generally the same as the name of the actual service, but not always. The filename will always be unique, which is why it should be used.

defaults

A list of default attributes to be set in the main configuration file (see *conffile* above).

defassignop_*attributeattribute!xinetd resource*

The assignment operator to be used in the configuration line for this default attribute. Default is `=`, but `-=` and `+=` can also be used.

defvalue_*attributeattribute!xinetd resource*

The value for the default attribute.

services

A list of services for which attributes and values will be defined through resources. Note that these services can be already defined (i.e. already have a configuration file in the directory specified by *basedir*) - in this case, attributes defined through resources will be added to the service definition and, in the event of conflicts, will override previously set values.

attributes_*serviceservice!xinetd resource*

A list of attributes that will be defined for each service listed in *services* above.

assignop_*service_attributeservice_attribute!xinetd resource*

The assignment operator to be used in the configuration line for this attribute. Default is =, but -= and += can also be used.

value_*service_attributeservice_attribute!xinetd resource*

The value for the attribute.

EXAMPLES

Here are some examples of how to use resources to control xinetd.

To set the default values in the main configuration file (see *conffile*), use resources in this way:

```
xinetd.defaults                instances log_type log_on_success \
                                   log_on_failure cps
xinetd.defvalue_instances      60
xinetd.defvalue_log_type       SYSLOG authpriv
xinetd.defvalue_log_on_success HOST PID
xinetd.defvalue_log_on_failure HOST
xinetd.defvalue_cps            25 30
```

This would result in the following defaults section being written to the xinetd.conf file:

```
defaults
{
      instances = 60
      log_type = SYSLOG authpriv
      log_on_success = HOST PID
      log_on_failure = HOST
      cps = 25 30
}
```

To specify a service entirely using LCFG resources, you would need something like the following:

```
xinetd.services                          myservice
xinetd.attributes_myservice              flags socket_type wait user \
                                           server log_on_failure
xinetd.value_myservice_flags             REUSE
xinetd.value_myservice_socket_type       stream
xinetd.value_myservice_wait              no
xinetd.value_myservice_user              root
xinetd.value_myservice_server            /usr/sbin/myserviced
xinetd.assignop_myservice_log_on_failure +=
xinetd.value_myservice_log_on_failure    USERID
```

This would result in a service definition file being created in the directory specified by the *confdir* resource. The file for the resources above would contain the following definition:

```
service myservice
{
    flags = REUSE
    socket_type = stream
    wait = no
    user = root
    server = /usr/sbin/myserviced
    log_on_failure += USERID
}
```

In addition to specifying a service in its entirety, it is possible to add new attributes to an existing service, or override values which have already been set, replacing the defaults in the service's configuration file. For example:

```
xinetd.services                        telnet
xinetd.attributes_telnet               log_on_failure only_from
xinetd.assignop_telnet_log_on_success +=
xinetd.value_telnet_log_on_success     USERID DURATION
xinetd.value_telnet_only_from          .localdomain.com
```

Thse resources would result in a new attribute `only_from` being defined for the telnet service and the new value for `log_on_success` would override the value in /etc/xinetd.d/telnet (pathname here is dependent on *basedir*).

SEE ALSO

xinetd(8), xinetd.conf(5), xinetd.log(5)

AUTHOR

Toby Blake <toby@inf.ed.ac.uk >

VERSION

0.99.7-1

Appendix C

Utilities

C.1 lcfglock

Lock/unlock component semaphore

SYNOPSIS

/usr/sbin/lcfglock [*options*] *component*

DESCRIPTION

This command is used by LCFG components to prevent multiple simultaneous executions of component methods.

OPTIONS

-b

When used in conjunction with **-u** this option forces the lock on the named component to be broken, even if it was owned by some other process.

-d *dir*

Use directory *dir* for lock files.

-D

Print debugging messages to stderr.

-n

Lock operations return immediately with exit status 2 if the semaphore is busy rather than waiting.

-p *pid*

Use *pid* as the process owning the semaphore. The default is is the process calling `lcfglock`.

-q

Quiet mode. exit silently when attempting to release non-existent locks, or to take already existing locks.

-t *secs*

Lock operations return immediately with exit status 2 if the semaphore is busy after waiting *secs* seconds.

-u

Unlock (rather than lock) the semaphore.

-v

Print messages when waiting for lock.

PLATFORMS

Redhat7, Redhat9, Solaris9

AUTHOR

Paul Anderson <dcspaul@inf.ed.ac.uk >

VERSION

1.1.23-1

C.2 lcfgmsg

Send messages to LCFG error/logging system

SYNOPSIS

/usr/sbin/lcfgmsg [*options*] *component message*

DESCRIPTION

This command is used by LCFG components and daemons to report error and log messages.

OPTIONS

-a

Send a SIGUSR2 to the client component requesting an acknowledgement be sent to the server.

-C *event*

Clear the named event log (delete the file).

-d

Send a Debug message.

-e

Send an Error message (non-fatal error).

-E *event*

Send an event message to the named event log.

-f

Send a Fail message (fatal error).

-i

Send an Info message. This message appears in the logfile and on the terminal.

-l

Send a message to the log file.

-n *tag*

Send a notification message to the monitoring system using ther given tag.

-o

Send an OK message. This message is reported to the terminal only.

-p

Advance progress bar.

-s

Start a progress bar.

-w

Send a warning message.

-x

End progress bar.

ENVIRONMENT VARIABLES

LCFG_MONITOR

If this is set to the name of a pipe, erorrs, warnings and monitoring information will be written to the named pipe.

LCFG_SYSLOG

If this is set to the name of a syslog facility, errors and warnings will be copied to syslog.

SEE ALSO

LCFG::Utils, lcfgutils, lcfg-ngeneric, LCFG::Component

PLATFORMS

Redhat7, Redhat9, Solaris9

AUTHOR

Paul Anderson <dcspaul@inf.ed.ac.uk >

VERSION

1.1.23-1

C.3 mkxprof

Make XML LCFG profile

DESCRIPTION

This command creates XML profiles from LCFG sources files. If source filenames are given on the command line, profiles will be generated for any host files listed explicitly, and for any which change because they depend on changes in one of the listed files.

If no files are specified, all source files (including headers, defaults and package lists) in the corresponding paths are examined, and any which have changed since the last run are recompiled.

mkxprof is normally run from the LCFG profile component.

SYNOPSIS

/usr/sbin/mkxprof [*options*] [*filename*..]

OPTIONS

-c *dir*

This directory is used to maintain caches of persistent state between invocations of mkxprof. It includes spanning map data, dependency information and status recrods. When running as root, the default is /var/lcfg/conf/server/cache. When running as any other user, no persistent state information is maintained unless this option is specified.

-C *component*

Error messages are passed to the log system for the named component.

-d

If this option is present, mkxprof runs as a daemon, polling or waiting for notifications of changed source files.

-D *flags*

This option enables debugging for the categories listed by the comma-separated list of *flags*. Flags may be prefixed with **+** or **-** to enable/disable specific categories. Possible flags are:

ack - Acknowledgements

assign - Resource assignments

changes - Profile changes

context - Contexts

cpp - CPP output

cppcmd - CPP commands

daemon - Daemon polling

defaults - Adding defaults

depend - Dependency generation

lock - Status DBM locking

mapchange - Changes in exported **resources**

maps - Spanning maps

meta - Meta-resource processing

mutate - Mutations

notify - Client notification

order - List sorting

packages - Packages

publish - Profile publication

ref - References

rsync - Rsync fetches

sources - Source files

validate - Validations

-E *path*

Search the (comma-separated) path for default files. The files must have an extension of **.def**. The string **%r** will be substituted with the value of the **profile.release** resource. The default is /usr/lib/lcfg/defaults/server.

-f *speclist*

The *speclist* is a (comma-separated) list of specifications of the form *dst*=*src*. **rsync** is used to copy each *src* to the corresponding *dst* before compiling the sources.

-F *path*

Search the (comma-separated) path for validation files. The default is /var/lcfg/conf/server/validation.

-h

If this option is present, status information, including all errors and warnings are stored for display on an HTML status page, rather than being printed to stdout. Normally, the CGI scripts **status** and *index* will be used to display this status information. The **-s** option can be used to automatically generate static HTML pages which do not require the CGI sripts, howver this is less flexible and results in slower compilations.

-H *path*

Search the (comma-separated) path for header files. The files must have an extension of **.h**. The default is /var/lcfg/conf/server/include,/usr/lib/lcfg/server/include.

-L *lockfiles*

The *lockfiles* argument is comma-separated list of full pathnames for lock files. In daemon mode, **mkxprof** will not compile source files while any of these lockfiles exist; the compilation will be deferred until the next poll, or notification. This provides a mechanism to allow several synchronized changes to be made to related files, in an atomic way.

-o *opts*

Additional options for the rsync command (see option **-f**). This can be used, for example, to specify included, or excluded files.

-N *fqdn*

The full name of the server to be used in status displays etc. By default, this is obtained from the **hostname** command, but an alias may be preferred for the server name.

-p *time*

When running as a daemon, this options species an interval to poll for changes to source files. It has the format: *time***h**|**m**|**s**[+*random***h**|**m**|**s**]. The random addition can be used to distribute server load.

-P *path*

Search the (comma-separated) path for package lists. The files must have an extension of **.pkgs** or **.rpms**. The string **%r** will be substituted with the value of the **profile.release** resource. The default is /var/lcfg/conf/server/packages.

-r

If this option is present, mkxprof adds a **derivation** attribute to each resource indicating the source files(s) and line number(s) at which the resource is defined.

-R

If this option is present, mkxprof rebuilds the dependency cache.

-s

This option generates static HTML pages containing status information. Normally, the **-h** option should be used instead.

-S *path*

Search the (comma-separated) path for source files. These files have no extension. The default is /var/lcfg/conf/server/source.

-v

Verbose.

-w *dir*

The root of the published web directory. Profiles are generated in the **profiles** subdirectory, and status reports are generated in the **status** subdirectory. The default is /var/lcfg/conf/server/web when running as root, and ./LCFG when running as any other user.

-W *flags*

This option enables warnings for the categories listed by the comma-separated list of *flags*. Possible flags are **ack**, **ambiguous**, **cache**, **client**, **components**, **context**, **files**, **mutate**, **ref**. Flags may be prefixed with **+** or **-** to enable/disable specific categories.

-x *file*

Write statistics records to the named file. For each compilation pass, a colon-separated record is written with the following fields:

```
Unix time at start of compilation pass
Unix time at end of compilation pass
Number of ACKs received during this pass
Number of acks discarded (superseded by later ones)
Number of files examined
Number of files changed (or explicitly specified)
Number of recompiled hosts
```

SIGNALS

When running in daemon mode, mkxprof will accept UDP notifications from clients on the service port **lcfgack** (default 733). These notifications contain the timestamp of the latest received profile which mkxprof will record in the status display.

A HUP signal causes the mkxprof daemon to re-examine the source files. This can be initiated remotely by **ssh** or by **om**, using the **run** or **rebuild** methods of the **profile** component.

An INT signal will terminate the daemon cleanly.

FILES

/var/lcfg/conf/server/cache

Contains profile caches and dependency information used internally by mkxprof.

/var/lcfg/tmp/server

Contains temporary files.

/var/lcfg/conf/server/web

Directory for profiles and status files. This directory should be published by a web server.

PLATFORMS

Redhat7, Redhat9, Solaris9

AUTHOR

Paul Anderson <dcspaul@inf.ed.ac.uk >

VERSION

2.1.64-1

C.4 qxprof

Query LCFG profile

SYNOPSIS

/usr/bin/qxprof [*options*] [*component*[.*resource*]] | [*resource*=*value*] ...

DESCRIPTION

This command queries the DBM file generated by rdxprof for resource values and information. If a component without a resource is specified, all resources with a non-empty value for that component are shown. Variable assignments specified on the command line overide any corresponding resource values.

OPTIONS

-a

Show all resources for the given component, even if they have a null value.

-d

Dump resources values to stdout. This is the default if **-e** and **-w** are not specified.

-e

The resources are printed in a format which is suitable for direct evaluation by the shell. This creates environment variables for all specified resources, prefixed with `LCFG_`*component*_. Note that you probably want to disable globbing (`set -f`) when evaluating the output form qxprof, otherwise unexpected shell expansions may occur.

-h *hostname*

Use resources for the specified host, rather than the current host. Note that this is only useful if the a DBM file for the specified host exists on the current machine; this will not normally be the case. **rdxprof** can however be used to fetch the profile for any machine and create the appropriate DBM file.

-i

Instead of reading resources from the profile, the resources are read from variables in the current environment, as created with the `-e` option.

-l

Load resources from profile. This is the default if neither **-i** or **-r** is specified.

-p *pfx*[,*pfx*]

The specified prefixes are used when creating shell variable names from resource names for exporting (or importing) resources into the environment. The first prefix is for variable names representing resource values, the second is for resource types. A **%s** in the prefix is replaced with the component name. The default values are: `LCFG_%S_,LCFGTYPE_%_`.

-r *file*

Read resources from the named file.

-v

Verbose. As well as the value of the resource, print out the derivation and type, if available.

-w *file*

Write resources to the named file.

FILES

/var/lcfg/conf/profile/dbm/*hostname*

The DBM file.

PLATFORMS

Redhat7, Redhat9, Solaris9

AUTHOR

Paul Anderson <dcspaul@inf.ed.ac.uk >

VERSION

1.1.23-1

C.5 rdxprof

Read XML LCFG profile

DESCRIPTION

This command optionally fetches an XML LCFG profile from a web server and converts it into a DBM file. If no hostname is specified, the profile for the local host is used.

rdxprof is normally run from the LCFG **client** component.

SYNOPSIS

/usr/sbin/rdxprof [*options*] [*hostname*]

OPTIONS

-a

If this option is present, rdxprof will send UDP acknowledgements on service port lcfgack (default 733) to the server(s) containing the timestamp of the last received profile.

-A *time*

When running as a daemon and sending server acknowledgements, this option specifies the minimum and maximum times between acknowledgements in the form *min***h**|**m**|**s**[+*max***h**|**m**|**s**]. Acknowledgements will never be sent faster than *min* apart, and will never be delayed for more than *max*.

-C *component*

Error messages are passed to the log system for the named component.

-d

If this option is present, rdxprof runs as a daemon, polling or waiting for notifications of changed profiles.

-D *flags*

This option enables debugging for the categories listed by the comma-separated list of *flags*. Possible flags are **ack**, **all**, **attrs**, **callbacks**, **context**, **changes**, **daemon**, **fetch**, **parse**, **rpms**. Flags may be prefixed with **+** or **-** to enable/disable specific categories.

-n

If this option is specified, then rdxprof will notify other components when their resources change by calling the method specified in the **client.reconfig_***component* resource.

-p *time*

When running as a daemon, this option species an interval to poll for new profiles. It has the format: *time***h**|**m**|**s**[+*random***h**|**m**|**s**]. The random addition can be used to distribute server load.

-r *prefix*

A prefix to be used for all pathnames. This is used by the profile component at install time when the root of the client filesystem is not the same as the current root.

-t *time*

The timeout interval for HTTP requests, in the form *time***h**|**m**|**s**.

-u *urls*

A comma-separated list of URLs for servers containing copies of the profile. If the URLs do not end in **.xml**, rdxprof will append *domain*/*hostname*/**XML/profile.xml**. rdxprof will then attempt to fetch new profiles from the specified URLs in random order. If this option is not present, the profile is assumed to already exist in /var/lcfg/conf/profile/xml. If any **file:** URLs are specified, they are tried before remote URLs. URLs staring with **none:** are ignored (this may be useful for passing to the client install method). A URL consisting simply of the string **file:** is assumed to refere to the default location of the local profile (this is useful during the install process).

-U *component method*

If this option is present, the component method *component method* will be called whenever any RPMs in the profile change. The *method* may be followed by any necessary options. This requires the **-n** option.

-v

Verbose.

-W *flags*

This option enables warnings for the categories listed by the comma-separated list of *flags*. Possible flags are **all**, **conflict**, **context**, **notify**, **error**, **fetch**, **parse**, **rpms**, **server**. Flags may be prefixed with **+** or **-** to enable/disable specific categories.

-x *path*

The the path of the directory containing the profile file (with the name profile.xml). If a client and server are both running on the same machine, it is useful to set this to the pathname of the profile directory used by the server. This allows the client to retrieve new profiles directly from the local disk, as they are generated by the server, without requiring a running web server.

SIGNALS

When not running in daemon mode, rdxprof will attempt to fetch the profile from the specified URL (if any) and rebuild the dbm file.

When running in daemon mode, a new profile will only be fetched if it is newer than the current profile. This will only be rebuilt into a new dbm file, if the profile is newer than the dbm file.

In daemon mode, UDP packets on service port **lcfg** (default 732) can be sent by mkxprof to initiate a poll for new profile. The following signals are also recognised:

HUP

This is equivalent to a notification packet from the server; it initiates a poll for a new profile.

USR1

This initiates a rebuild of the dbm file, if the profile, or the context has changed. This signal is sent by the client component after changing the context.

POLL

The initiates a rebuild of the dbm file regardless of any changes to the profile or the context.

USR2

This requests an acknowledgement to be sent to the server.

INT

Requests server termination.

FILES

/var/lcfg/conf/profile/xml

The directory containing the profiles.

/var/lcfg/conf/profile/dbm

The directory containing the generated DBM files.

/var/lcfg/conf/profile/context

The directory containing the context files.

/var/lcfg/conf/profile/rpmcfg

The directory containing the client RPM configuration files.

PLATFORMS

Redhat7, Redhat9, Solaris9

AUTHOR

Paul Anderson <dcspaul@inf.ed.ac.uk >

VERSION

2.1.35-1

C.6 shiftpressed

Detect if shift key pressed

SYNOPSIS

/usr/sbin/shiftpressed

DESCRIPTION

This command tests if the shift key is pressed on the console connected to the stdin. It returns an exit status of 0 if the key is pressed, 2 if it is not, and 1 if the stdout is not a console or the state cannot be determined for some other reason.

PLATFORMS

Redhat7, Redhat9

AUTHOR

Paul Anderson <dcspaul@inf.ed.ac.uk >

VERSION

1.1.23-1

C.7 sxprof

Substitute LCFG resource values in template

SYNOPSIS

/usr/bin/sxprof [*options*] *component* [*template* [*target-file*]] | [*var=value*] ...

DESCRIPTION

Substitute LCFG resources from the specified component into the given template, to generate the named target file. Variable assignments specified on the command line overide any corresponding resource values.

OPTIONS

-B

Do not create backup files. Normally backup files are created with an extension of "~".

-d

Dummy run. Do not change target files, but still report which files would have changed, and set exit status accordingly.

-h *hostname*

Use resources for the specified host, rather than the current host. Note that this is only useful if the a DBM file for the specified host exists on the current machine; this will not normally be the case. **rdxprof** can however be used to fetch the profile for any machine and create the appropriate DBM file.

-i

Instead of reading resources from the profile, the resources are read from variables in the current environment, as created with the `-e` option of **qxprof**.

-l

Load resources from profile (default).

-L *delimiter*

Set the left delimiter used for substituted expresions (default <**%**). This is Perl regexp and sxprof wil fail if meta-characters are not correctly escaped.

-p *pfx*[,*pfx*]

The specified prefixes are used when creating shell variable names from resource names for importing resources from the environment. The first prefix is for variable names representing resource values, the second is for resource types. A **%s** in the prefix is replaced with the component name. The default values are: `LCFG_%S_,LCFGTYPE_%_`.

-r *file*

Read resources from named file rather than profile.

-R *delimiter*

Set the right delimiter used for substituted expresions (default **%** >). This is Perl regexp and sxprof wil fail if meta-characters are not correctly escaped.

-t

The component resource `template` is expected to contain a list of tags specifying a template to be processed. The resources `tsrc_`*tag* and `tdst_`*tag* should contain the template source and target filenames. These templates are processed before any templates specified on the command line.

-v

Verbose.

EXIT STATUS

1. 0

   No target files have been changed.

2. 1

   Error.

3. 2

   At least one of the target files has changed.

TEMPLATE LANGUAGE

See the manual page for the Perl module `LCFG::Template`.

FILES

/var/lcfg/conf/profile/dbm/*hostname*

The DBM file.

/usr/share/doc/lcfg-utils-1.1.23/EXAMPLE

An example template.

PLATFORMS

Redhat7, Redhat9, Solaris9

AUTHOR

Paul Anderson <dcspaul@inf.ed.ac.uk >

VERSION

1.1.23-1

Appendix D

Solaris Jumpstart Scripts

D.1 The start script

```
#!/bin/sh
# set hostname to non fully qualified.
DAIHOSTNAME=‘uname -n‘
if echo $DAIHOSTNAME | grep ".ed.ac.uk" > /dev/null ; then
  DAIHOSTNAME=‘echo $DAIHOSTNAME | cut -f1 -d.‘
  hostname $DAIHOSTNAME
  SI_HOSTNAME=$DAIHOSTNAME
  export SI_HOSTNAME
fi
TZ=GB-Eire export TZ
umask 022

# NFS mount LCFG utilities
# things this must include:
# - LCFG perl modules -- added to perl path
# - LCFG client (inc rdxprof)
# - tsort, cpp, gunzip (used by updatepkgs - not here)
LCFGNFS=harpy:/export/lcfg/image
LCFGMNT=/tmp/lcfg
PROFILEURL=http://tattie.inf.ed.ac.uk/profiles/
mkdir $LCFGMNT
mount -F nfs $LCFGNFS $LCFGMNT
PATH=$PATH:$LCFGMNT/bin:$LCFGMNT/sbin:$LCFGMNT/usr/bin:$LCFGMNT/usr/sbin
export PATH
# download LCFG profile (adding lcfg perl module directory to module
# search path)
PERL5LIB=$LCFGMNT/usr/perl5/5.6.1/lib/sun4-solaris-64int
PERL5LIB=$LCFGMNT/usr/perl5/5.6.1/lib:$PERL5LIB
PERL5LIB=$LCFGMNT/usr/perl5/site_perl/5.6.1/sun4-solaris-64int:$PERL5LIB
PERL5LIB=$LCFGMNT/usr/perl5/site_perl/5.6.1:$PERL5LIB
PERL5LIB=$LCFGMNT/usr/perl5/site_perl:$PERL5LIB
export PERL5LIB
rdxprof -u $PROFILEURL

# create initial jumpstart profile
echo "install_type initial_install" > $SI_PROFILE
echo "system_type standalone" >> $SI_PROFILE
# install the core cluster - everything else is installed in
# lcfg_setup post-install
echo "cluster SUNWCreq" >> $SI_PROFILE
```

```
# add fstab stuff - partitioning / filesys
echo "partitioning explicit" >> $SI_PROFILE
for disk in ‘qxprof fstab.disks | sed s/.*=//‘
do
for slice in ‘qxprof fstab.partitions_$disk | sed s/.*=//‘
do
echo "filesys $slice ‘qxprof fstab.size_$slice                       | sed s/.*=//‘ ‘qxprof fstab.
done
done
```

D.2 The finish script

```
#!/bin/sh
# We’re read directly by the suninstall script.  Execute all in subshell
# We don’t want to influence our parent.
(
#  mountpoint of / for the new system
newroot=/a
export newroot

# set a nice default umask
umask 022

#  location of files to install
install=$SI_CONFIG_DIR/install

# copy the LCFG installation stuff into /a/etc/rc2.d so it gets
# run on first reboot - do additional software installation there
cp $install/files/lcfg_setup $newroot/etc/rc2.d/S80lcfg_setup
chmod 755 $newroot/etc/rc2.d/S80lcfg_setup

echo installation complete
# Subshell end
)
```

Appendix E

Standard Symbols

E.1 Symbols defined in os.mk

AR	The GNU ar program
AWK	The GNU awk program
CC	The GNU C compiler
DARWIN_ONLY	Set to the comment symbol (#) except on Darwin (OS X)
EGREP	The GNU egrep program
ENCODING	Perl commands to set byte encoding for scripts and input files
GREP	The GNU grep program
INITDIR	The directory for init scripts
INSTALL	The GNU install program
LCFGOS	The OS name, as returned by uname -s
LIBMANSECT	The manual page section for libraries
LINUX_ONLY	Set to the comment symbol (#) except on Linux
MAKE	The GNU make program
MANSECT	The manual page section for admin commands
OS_RELEASE	The OS release
OS_VERSION	The OS version
PERL	The pathname of the perl interpreter
PERL_INST	The directory set to use for Perl installations (”site” or ”vendor”)
PERL_VERSION	The perl version
RSYNC	The location of rsync
SED	The GNU sed program
SHELL	The bash shell
SOLARIS_ONLY	Set to the comment symbol (#) except on Solaris
SORT	The GNU sort program
TAR	The GNU tar program
TARHASNOT	True if tar has no T option

E.2 Symbols defined in `lcfg.mk`

LCFGBIB	Directory for BIB files.
LCFGBIBURL	URL for BIB files.
LCFGBIN	Directory for user binaries.
LCFGCLIENTDEF	Directory for default resource files used by client
LCFGCONF	Directory for generated files to be preserved between object runs. Files are normally prefixed with the module name, or stored in subdirectories with the same name as the module.
LCFGCONFIGMSG	A string giving the version of buildtools
LCFGDATA	Directory for templates and other fixed configuration files. Files are normally prefixed with the module name, or stored in subdirectories with the same name as the module.
LCFGDEF	Deprecated (use LCFGSERVERDEF)
LCFGDOC	Base directory for documentation.
LCFGHTML	Directory for HTML files.
LCFGHTMLURL	URL for HTML files.
LCFGLIB	Base directory for read-only files.
LCFGLOCK	Directory for lock files.
LCFGLOG	Directory for log files.
LCFGMAN	Base directory for man pages.
LCFGOM	Location of ”om” program
LCFGPDF	Directory for PDF files.
LCFGPDFURL	URL for PDF files.
LCFGPERL	Directory for Perl modules. Normally in the subdirectory `LCFG::`.
LCFGPOD	Directory for POD files.
LCFGROTATED	Directory for log rotate files.
LCFGSBIN	Directory for system binaries.
LCFGSERVERDEF	Directory for default resource files used by server
LCFGSTATUS	Directory for status files.
LCFGTMP	Directory for temporary files (may be deleted when objects are not running). Files are normally prefixed with the module name, or stored in subdirectories with the same name as the module. Components should not store temporary files in system tmp directories.
LCFGURL	Base URL for documentation.
TESTCONF	Config directory used in test environment
TESTING	Set to ”yes” to use test environment
TESTLOCK	Lock directory used in test environment
TESTLOG	Logfile used in test environment
TESTPERLV	Perl assignment to set up test environment
TESTPID	PID file used in test environment
TESTRES	Resource file used in test environment
TESTROOT	Root of test directory structure for test environment
TESTROTATE	Logrotate directory used in test environment
TESTRUN	Run file used in test environment
TESTSHELLV	Shell assignments to setup test environment
TESTSRES	Saved resource file used in test environment
TESTSTATUS	Statusfule used for testing

E.3 Symbols defined in site.mk

dice The site.mk file contains site-specific definitions. Under DICE, these are:

DICEBIB	Directory for BIB files.
DICEBIBURL	URL for BIB files.
DICEBIN	Directory for user binaries.
DICEDOC	Base directory for documentation.
DICEHTML	Directory for HTML files.
DICEHTMLURL	URL for HTML files
DICELIB	Base directory for read-only files.
DICEMAN	Directory for man pages.
DICEPDF	Directory for PDF files.
DICEPDFURL	URL for PDF files.
DICEPERL	Directory for Perl modules. Normally in the subdirectory DICE::.
DICEPOD	Directory for POD files.
DICESBIN	Directory for system binaries.
DICEURL	Base URL for documentation

Appendix F

Perl Modules

F.1 LCFG::Component

Perl module for LCFG Generic component

DESCRIPTION

This module provides a superclass for creating LCFG components in Perl.

Components should subclass LCFG::Component, create a new instance of the class, and call the **Dispatch** method to excute the component method specified in the command line arguments. The LCFG component **perlex** shows how this is used in practice.

LCFG::Component attempts to provide an identical functionality to the shell generic component **ngeneric**.

FUNCTIONS

Equivalent Perl functions are provided for all the method functions described in **lcfg-ngeneric**. A hash of resource values is passed as the first argument to each function, rather than passing the resources in the environment.

RESOURCES, LOCKING and LOG ROTATING

See the documentation for the **ngeneric** component.

VARIABLES

All the variables described for **lcfg-ngeneric** have equivalent instance variables in the LCFG::Component class.

AD-HOC METHODS

Methods with names of the form `Method`_*methodname*, will be automatically called by the Dispatch function. Ad-hoc methods should arrange to call the **Lock** function if appropriate to prevent simultaneous method calls.

SEE ALSO

lcfg-ngeneric

The shell generic functions.

lcfg-perlex

An example component.

LCFG::Template

The template processor.

LCFG::Resources

The resource handling functions.

PLATFORMS

Redhat7, Redhat9, Solaris9

AUTHOR

Paul Anderson <dcspaul@inf.ed.ac.uk >

VERSION

1.1.23-1

F.2 LCFG::Inventory

Fetch and parse XML inventory

SYNOPSYS

use LCFG::Inventory;

```
# Fetch and Parse LCFG inventory from server

$inv = new LCFG::Inventory
   ( URL => "http://blah",           # URL
     CACHE => "/foo/mycache",        # Persistent cache
     DEBUG => 1,                     # Debugging
     FORCE => 1,                     # Force refresh the cache
     NOFETCH => 1                    # Use cache copy only
   );

# Return list of FQDNS in inventory

@hosts = $inv->Hosts();

# Return inventory fields for given FQDN

$hash = $inv->Lookup("foo.bar.com");

# Return hash of meta-information about inventory

$hash = $inv->Meta();
```

DESCRIPTION

When a new LCFG::Inventory object is created, the XML inventory information is fetched from the specified URL, parsed and cached in a local file. The **Hosts()** function will return a list of FQDNs for all the hosts in the inventory, and the **Lookup()** function returns a hash of the inventory fields for a given FQDN.

If an explicit CACHE option is given, the named file will be used to store a persistent cache which will only be refreshed when the remote XML changes. The FORCE option can be used to force the refresh of a persistent cache, and the NOFETCH option can be used to force the use of the local cache without checking the remote copy.

EXAMPLE

/usr/share/doc/lcfg-inventory-1.1.3/example

PLATFORMS

Redhat7, Redhat9, Solaris9

AUTHOR

Paul Anderson <dcspaul@inf.ed.ac.uk >

VERSION

1.1.3-1

F.3 LCFG::Resources

Load and save LCFG resources

SYNOPSYS

use LCFG::Resources;

```
# Load resources for named resources from adaptor profile
$res = LCFG::Resources::Load($hostname,$rspec1,$rspec2,...);

# Dump resources for named resources to stdout
LCFG::Resources::Dump($res,$verbose,$all,$rspec1,$rspec2,...);

# Dump named resources as shell assigments
LCFG::Resources::Export($res,$rspec1,$rspec2,...);

# Load resource for named resources from environment
$res = LCFG::Resources::Import($rspec1,$rspec2,...);

# Write named resources to file
LCFG::Resources::WriteFile($file,$res,$rspec1,$rspec2,...);

# Read named resources from file
$res = LCFG::Resources::ReadFile($file,$rspec1,$rspec2,...);

# Parse resource values from arguments
$res = LCFG::Resources::Parse($default,"res1=val1","res2=val2",...);

# Merge resource structures
$res = LCFG::Resources::Merge($res1,$res2,...);

# Set prefixes to be used for environment variables
LCFG::Resources::SetPrefix($value_prefix,$type_prefix);
```

DESCRIPTION

In the above, *rspec* has the form *component.resource* or simply *component* which refers to all resources in the specified component.

The **Parse** routine accepts qualified, or unqualified resource names. The *default* component is assumed for unqualified resource names.

SetPrefix defines the prefixes attached to resource names when the values and types are exported or imported from the environment. **%s** in the prefix strings is replaced by the name of the corresponding component. The defaults are `LCFG_%S_` and `LCFGTYPE_%s_`.

The *res* structures have the following form:

```
{
  'foo' => {
             'resource1' => {
                              VALUE => value,
                              TYPE => type,
```

```
                                  DERIVE = > derivation,
                                  CONTEXT => context
                                },
                 'resource2' => {
                                  VALUE => value,
                                  TYPE => type,
                                  DERIVE = > derivation,
                                  CONTEXT => context
                                },
                 ......
               }

      'bar' => {
                 'resource1' => {
                                  VALUE => value,
                                  TYPE => type,
                                  DERIVE = > derivation,
                                  CONTEXT => context
                                },
                 'resource2' => {
                                  VALUE => value,
                                  TYPE => type,
                                  DERIVE = > derivation,
                                  CONTEXT => context
                                },
                 ......
               }
      .......
 }
```

All routines return `undef` and set the variable `$@` on error.

PLATFORMS

Redhat7, Redhat9, Solaris9

AUTHOR

Paul Anderson <dcspaul@inf.ed.ac.uk >

VERSION

1.1.23-1

F.4 LCFG::Template

Substitute LCFG resources in template

SYNOPSYS

```
use LCFG::Template;


# Load resources for 'foo' and 'bar' from adaptor profile
$result = LCFG::Template::Substitute($template,$target,$mode,@res);


# Set delimiters
LCFG::Template::Delimiters($left,$right);
```

DESCRIPTION

This routine takes the name of a template file and substitutes LCFG resource values into the template from the given list of resource tables. The return status indicates whether the target file has been changed by the operation (1) or not (0). The `mode` option can be used to specify the following bit flags: if <mode >&1 is non-zero, then then the file is never modified, but the return status indicates whether or not it would have been. If <mode >&2 is non-zero, then no backup files are created.

The resources tables are in the same format as generated by the LCFG::Resources module; note that this includes the name of the component at the top level of the structure:

```
{
  'mycomp' => {
             'resource1' => {
                               VALUE => value,
                               TYPE => type,
                               DERIVE = > derivation,
                               AU = > authors,
                               CONTEXT => context
                             },
             'resource2' => {
                               VALUE => value,
                               TYPE => type,
                               DERIVE = > derivation,
                               AU = > authors,
                               CONTEXT => context
                             },
             ......
           }
```

If an error occurs, then the routine returns `undef` and the variable `$@` contains the error message.

TEMPLATE LANGUAGE

The following constructions are supported in the template:

<%resource% >

Substitute the value of the named resource. The resource name my be preceeded by a `#` in which case the "derivation" of the resource will be substituted instead of the value. This can be usefully used to generate comments in the generated configuration file indicating the source of the various parameters. The delimiters

<%{%> and <%}%> (see below) are useful when substituting derivations in comments to prevent a reconfiguraion being flagged if only the derivations (and not the values) change.

Note that the LCFG client component will only notify components of changes to the value of resources – if only derivations change, then the component is not automatically reconfigured, and values of substituted derivations may be out of date.

<%if: *expr*%> *text* **<%else:%>** ***text*** **<%end:%>**

If the *expr* is non-null, then substitute the first text, otherwise substitute the second text. The *else* part is optional.

<%perl: *expression*%>

Substitute the result of the Perl *expresssion*.

<%shell: *command*%>

Substitute the result of the Shell *command*.

<%ifdef: *resource*%> *text* **<%else:%>** ***text*** **<%end:%>**

If the *resource* is defined, then substitute the first text, otherwise substitute the second text. The *else* part is optional.

<%for: *var*=*expr*%> ***text*** **<%end:%>**

Substitute one copy of the specified *text* for each item in the space-separated list *expr*. During substitution of the text, the value of the variable *var* may be referenced as <%*var*%>. (Any resource with the same name as var will be inaccessible during the scope of the statement).

<%set: *var*=*expr*%>

Set a global variable to the given value. The global variable can be accessed as <%*var*%> at any subsequent point in the program. Any resource with the same name will be inacessible.

<%include: *filename*%>

Include the contents of the specified template file, evaluating it in the current context.

<%\%>

Delete any following white space. This allows complex template expressions to span multiple lines, while still generating output on a single line.

<%/*%> ... <%*/%>

Text between these delimiters is treated as a comment in the template and is not copied to the output file.

<%{%> ... <%}%>

Text between these delimiters is treated as insignificant. The text is still copied to the output file (evaluating any expressions), but changes to this text are not sufficient for the return status to indicate that the file has changed. This is useful for placing changing comments in the output (for example indicating the generation date) without triggering reconfiguration of the component unless something significant has changed. Eg:

```
#<%{%> Generated on <%shell: date%> <%}%>
```

All the above elements except *var* may contain nested statements.

PLATFORMS

Redhat7, Redhat9, Solaris9

AUTHOR

Paul Anderson <dcspaul@inf.ed.ac.uk>

VERSION

1.1.23-1

F.5 LCFG::Utils

LCFG Utility Functions

SYNOPSIS

```
# Select fd for message output
SetOutput( $fd )

# Send Debug Message
Debug( $component, $msg )

# Send Info message (log file and terminal)
Info( $component, $msg )

# Send Log Message (logfile only)
Log( $component, $msg )

# Send Log Message with sepcial prefix
LogPrefix( $component, $pfx, $msg )

# Send OK Message (terminal only)
OK( $component, $msg )

# Send Warning Message
Warn( $component, $msg )

# Send Error Message (non fatal)
Error( $component, $msg )

# Send Fail Message (fatal error)
Fail( $component, $msg )

# Send Monitoring Message
Notify( $component, $tag, $msg )

# Send Message to named event log
Event( $component, $event, $msg )

# Clear event log
ClearEvent( $component, $msg )

# Start a Progress Bar
StartProgress( $component, $msg )

# Advance Progress Bar
Progress()

# End Progress Bar
EndProgress()

# Signal client component to acknowledge server
Ack()

# Detect if shift key pressed
ShiftPressed()
```

DESCRIPTION

These routines are Perl bindings for the LCFG utility routines in **lcfgutils**.

ENVIRONMENT VARIABLES

LCFG_MONITOR

If this is set to the name of a pipe, erorrs, warnings and monitoring information will be written to the named pipe.

LCFG_SYSLOG

If this is set to the name of a syslog facility, errors and warnings will be copied to syslog.

SEE ALSO

LCFG::Utils, lcfgmsg, lcfg-ngeneric, LCFG::Component

PLATFORMS

Redhat7, Redhat9, Solaris9

AUTHOR

Paul Anderson <dcspaul@inf.ed.ac.uk >

VERSION

1.1.23-1

Appendix G

C Libraries

G.1 lcfgutils

C library of LCFG utility routines.

SYNOPSIS

```
/* Set file descriptor for output (default stderr) */
void LCFG_SetOutput( FILE *fp )

/* Send Debug Message */
void LCFG_Debug( char *component, char *msg )

/* Send Info message (log file and terminal) */
void LCFG_Info( char *component, char *msg )

/* Send Log Message (logfile only) */
void LCFG_Log( char *component, char *msg )

/* Send Log Message with prefix */
void LCFG_Log( char *component, char *pfx, char *msg )

/* Send OK Message (terminal only) */
void LCFG_OK( char *component, char *msg )

/* Send Warning Message */
void LCFG_Warn( char *component, char *msg )

/* Send Error Message (non fatal) */
void LCFG_Error( char *component, char *msg )

/* Send Fail Message (fatal error) */
void LCFG_Fail( char *component, char *msg )

/* Send Monitoring Message */
void LCFG_Notify( char *component, char *tag, char *msg )

/* Send Message to named event log */
void LCFG_Event( char *component, char *event, char *msg )

/* Clear event log */
void LCFG_ClearEvent( char *component, char *msg )

/* Start a Progress Bar */
int LCFG_StartProgress( char *component, char *msg )

/* Advance Progress Bar */
int LCFG_Progress( void )

/* End Progress Bar */
int LCFG_EndProgress( void )

/* Signal client component to acknowledge server */
void LCFG_Ack( void )

/* Detect if shift key pressed (0=no, 1=yes, -1=don’t know) */
int LCFG_ShiftPressed( void )
```

DESCRIPTION

These routines are used by the LCFG generic component for reporting log and error information. Daemons can use the same routines for reporting so that error message are passed to the status and monitoring systems.

ENVIRONMENT VARIABLES

LCFG_MONITOR

If this is set to the name of a pipe, erorrs, warnings and monitoring information will be written to the named pipe.

LCFG_SYSLOG

If this is set to the name of a syslog facility, errors and warnings will be copied to syslog.

SEE ALSO

LCFG::Utils, lcfgmsg, lcfg-ngeneric, LCFG::Component

PLATFORMS

Redhat7, Redhat9, Solaris9

AUTHOR

Paul Anderson <dcspaul@inf.ed.ac.uk >

VERSION

1.1.23-1

Appendix H

Code Examples

H.1 Example Shell Component

This section includes default file and code for the example Shell component. Note that the code is shown after processing and substitution with the `lcfg-buildtools` (11). For an example of how the code would appear before processing, see appendix I.

H.1.1 Resource Defaults

```
/*
 * LCFG example component : default resources
 *
 * Paul Anderson <dcspaul@inf.ed.ac.uk>
 * Version: 1.1.4 02/04/04 10:34 (Schema 1)
 *
 *  ** Generated file : do not edit **
 *
 */

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

schema 1
server foo.bar.com
template /usr/lib/lcfg/conf/example/template
configfile /var/lcfg/conf/example/config
```

H.1.2 Example Component

```
#!/bin/bash
########################################################################
#
# Example LCFG Component
#
# Paul Anderson <dcspaul@inf.ed.ac.uk>
# Version 1.1.4 : 02/04/04 10:34
#
#  ** Generated file : do not edit **
#
########################################################################

 . /usr/lib/lcfg/components/ngeneric

###########################################################################
Configure() {
###########################################################################

  # Use sxprof to substitute the configuration parameters from the
  # environment into the template.
  /usr/bin/sxprof -i $_COMP $LCFG_example_template \
                           $LCFG_example_configfile

  # Was anything changed? Or did the substitution fail?
  status=$?; [ $status = 2 ] && LogMessage "configuration changed"
  [ $status = 1 ] && Fail "failed to create config file (see logfile)"

  # At this point, we should check if the daemon is running, and
  # if so notify it of any changes (if necessary)
}

###########################################################################
Start() {
###########################################################################

  # Start daemon here
  # Daemon "SOME COMMAND TO RUN MY DAEMON"
  return;
}

###########################################################################
Stop() {
###########################################################################

  # Stop daemon here
  return;
}

###########################################################################
# Dispatch methods
###########################################################################

Dispatch "$@"
```

H.2 Example Perl Component

This section includes default file and code for the example Perl component. Note that the code is shown after processing and substitution with the `lcfg-buildtools` (11). For an example of how the code would appear before processing, see appendix I.

H.2.1 Resource Defaults

```
/*
 * LCFG example component in Perl : default resources
 *
 * Paul Anderson <dcspaul@inf.ed.ac.uk>
 * Version: 1.1.3 07/08/03 16:47 (Schema 1)
 *
 *  ** Generated file : do not edit **
 *
 */

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

schema 1
server foo.bar.com
```

H.2.2 Perlex Component

```
#!/usr/bin/perl
#######################################################################
#
# Example LCFG Component in Perl
#
# Paul Anderson <dcspaul@inf.ed.ac.uk>
# Version 1.1.3 : 07/08/03 16:47
#
# ** Generated file : do not edit **
#
#######################################################################

use bytes; use open IO => ':bytes';
package LCFG::PerlEx;
@ISA = qw(LCFG::Component);

use strict;
use LCFG::Component;

##########################################################################
# Resource variables
##########################################################################

my $server = undef;

##########################################################################
sub Configure($$@) {
##########################################################################

  my $self = shift;
  my $res = shift;
  my @args = @_;

  ########################################################################
  # We illustrate two different cases here. Normally, you wouldn't
  # use both together:
  ########################################################################

  $server = $res->{'server'}->{VALUE};

  ########################################################################
  # (1) Firstly, we recreate a configuration file when we get a reconfigure
  # call. Normally, this would be used if you have no daemon, or if
  # your daemon is a separate program.
  ########################################################################

  my $status = LCFG::Template::Substitute
    ( '/usr/lib/lcfg/conf/perlex/template',
      '/var/lcfg/conf/perlex/config', 0, $res );

  unless (defined($status)) {
    $self->LogMessage($@);
    $self->Fail( "failed to create config file (see logfile)");
  }

  $self->LogMessage("configuration changed") if ($status==1);

  # At this point, we should check if the daemon is running, and
```

```
  # if so notify it of any changes (if necessary)

  ########################################################################
  # (2) Secondly, if we are writing our own daemon which runs as
  # a fork of this component code, then we use this routine to signal
  # the daemon to reload its resources
  ########################################################################

  $self->ConfigureDaemon($res,@args);
}

##########################################################################
sub Start($$@) {
##########################################################################

  my $self = shift;
  my $res = shift;
  my @args = @_;

  ########################################################################
  # Use this routine to start a daemon running as a fork of the
  # current code. This invokes the DaemonStart() routine.
  ########################################################################

  $self->StartDaemon($res,@args);

  ########################################################################
  # If you want to run an external daemon program, you should start
  # it here and record the PID somewhere so you can stop it later
  ########################################################################

  ########################################################################
  # If you don’t have a daemon, you don’t need a Start() routine
  # at all.
  ########################################################################
}

##########################################################################
sub Stop($$@) {
##########################################################################

  my $self = shift;
  my $res = shift;
  my @args = @_;

  ########################################################################
  # Use this routine to signal a daemon running as a fork of the
  # current code. This invokes the DaemonStop() routine.
  ########################################################################

  $self->StopDaemon($res,@args);

  ########################################################################
  # If you want to run an external daemon program, you should have
  # saved the PID in the Start() routine, so you can kill it here.
  # You probably want to wait here until you are satisfied that the
  # daemon really has stopped.
  ########################################################################

  ########################################################################
```

```
  # If you don’t have a daemon, you don’t need a Stop() routine
  # at all.
  ########################################################################
}

######################################################################
sub DaemonConfigure($$@) {
######################################################################

  my $self = shift;
  my $res = shift;
  my @args = @_;

  # This gets called * AT INTERRUPT TIME * in the daemon process
  # when any resources have changed. Only use this if you are
  # writing your daemon code as a fork of this component code.

  $self->LogMessage("daemon reconfigured: @args");

  $server = $res->{’server’}->{VALUE};
}

######################################################################
sub DaemonStop($$@) {
######################################################################

  my $self = shift;
  my $res = shift;
  my @args = @_;

  # This gets called * AT INTERRUPT TIME * in the daemon process
  # when the component is stopped. Only use this if you are
  # writing your daemon code as a fork of this component code.

  $self->LogMessage("daemon stopped: @args");
  exit(0);
}

######################################################################
sub DaemonStart($$@) {
######################################################################

  my $self = shift;
  my $res = shift;
  my @args = @_;

  # This is the main daemon loop.
  # Normally, this will not exit. It will be terminated by
  # an INT signal which invokes a call to DaemonStop().

  $self->LogMessage("daemon started: version 1.1.3 - @args");

  while (1) {
    $self->LogMessage("Hello World: server=$server");
    sleep(10);
  }
}

##########################################################################
# Dispatch methods
```

```
##########################################################################

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

Appendix I

Buildtools Examples

The following examples show source files from the `lcfg-example` component, as stored in the CVS – i.e. before processing and substitution by the `lcfg-buildtools` (11). For an example of how the code would appear before processing, see appendix H.1.

I.1 Sample config.mk for LCFG Component

```
COMP=example
NAME=lcfg-$(COMP)
DESCR=An Example LCFG component
V=1.1.4
R=1
SCHEMA=1
VERSION=$(V)
GROUP=LCFG
AUTHOR=Paul Anderson <dcspaul@inf.ed.ac.uk>
PLATFORMS=Redhat7, Redhat9, Solaris9

CONFIGDIR=$(LCFGCONF)/$(COMP)

MANDIR=$(LCFGMAN)/man$(MANSECT)

DATE=02/04/04 10:34
TARFILE=lcfg-example-1.1.4.src.tgz
PROD=
DEV=#
```

I.2 Sample Makefile for LCFG Component

```
####################################################################
# Distribution Makefile
####################################################################

.PHONY: configure install clean

all: configure

include buildtools.mk

####################################################################
# Configure
####################################################################

configure: $(COMP) $(COMP).def $(COMP).pod $(NAME).$(MANSECT)             template

####################################################################
# Install
####################################################################

install: configure
@echo installing ...
@mkdir -p $(PREFIX)$(LCFGCOMP)
@mkdir -p $(PREFIX)$(LCFGPOD)
@mkdir -p $(PREFIX)$(LCFGSERVERDEF)
@mkdir -p $(PREFIX)$(LCFGCLIENTDEF)
@mkdir -p $(PREFIX)$(MANDIR)
@mkdir -p $(PREFIX)$(LCFGDATA)/$(COMP)
@mkdir -p $(PREFIX)$(CONFIGDIR)
@$(INSTALL) -m 0555 $(COMP) $(PREFIX)$(LCFGCOMP)/$(COMP)
@$(INSTALL) -m 0555 template $(PREFIX)$(LCFGDATA)/$(COMP)/template
@$(INSTALL) -m 0444 $(NAME).$(MANSECT)                   $(PREFIX)$(MANDIR)/$(NAME).$(MANSECT)
@$(INSTALL) -m 0444 $(COMP).pod $(PREFIX)$(LCFGPOD)/$(COMP).pod
@$(INSTALL) -m 0444 $(COMP).def                   $(PREFIX)$(LCFGSERVERDEF)/$(COMP)-$(SCHEMA).def
@$(INSTALL) -m 0444 $(COMP).def                   $(PREFIX)$(LCFGCLIENTDEF)/$(COMP)-$(SCHEMA).def

####################################################################
# Cleanup
####################################################################

clean::
@echo cleaning $(NAME) files ...
@rm -f $(COMP) $(COMP).pod $(COMP).def $(NAME).$(MANSECT)
```

I.3 Sample Source for LCFG Component

```
#!@SHELL@
#######################################################################
#
# Example LCFG Component
#
# @AUTHOR@
# Version @VERSION@ : @DATE@
#
# @MSG@
#
#######################################################################

@TESTSHELLV@ . @LCFGCOMP@/ngeneric

##########################################################################
Configure()
##########################################################################

  # Use sxprof to substitute the configuration parameters from the
  # environment into the template.
  @LCFGBIN@/sxprof -i $_COMP $LCFG_example_template                              $LCFG_examp

  # Was anything changed? Or did the substitution fail?
  status=$?; [ $status = 2 ] && LogMessage "configuration changed"
  [ $status = 1 ] && Fail "failed to create config file (see logfile)"

  # At this point, we should check if the daemon is running, and
  # if so notify it of any changes (if necessary)


##########################################################################
Start()
##########################################################################

  # Start daemon here
  # Daemon "SOME COMMAND TO RUN MY DAEMON"
  return;


##########################################################################
Stop()
##########################################################################

  # Stop daemon here
  return;


##########################################################################
# Dispatch methods
##########################################################################

Dispatch "$@"
```

I.4 Sample POD for LCFG Component

```
=head1 NAME

example - An example LCFG component

=head1 DESCRIPTION

This component is an example only.

=head1 RESOURCES

=over 4

=item B<server>

An example resource which gets substituted into the configuration file.

=back

=head1 PLATFORMS

Redhat7, Redhat9, Solaris9

=head1 AUTHOR

Paul Anderson <dcspaul@inf.ed.ac.uk>
```

I.5 Sample specfile for LCFG Component

```
Summary: @DESCR@
Name: @NAME@
Version: @V@
Vendor: University of Edinburgh
Release: @R@
Copyright: GPL
Group: @GROUP@/Components
Source: @TARFILE@
BuildArch: noarch
BuildRoot: /var/tmp/%name-build
Packager: @AUTHOR@
Requires: lcfg-ngeneric

%description
An example LCFG component.
@LCFGCONFIGMSG@

%prep
%setup

%build
@MAKE@

%install
rm -rf $RPM_BUILD_ROOT
@MAKE@ PREFIX=$RPM_BUILD_ROOT install

%postun
[ $1 = 0 ] && rm -f @LCFGROTATED@/@NAME@
exit 0

%files
%defattr(-,root,root)
%doc ChangeLog README README.BUILD
%doc @LCFGMAN@/man@MANSECT@/*
%doc @LCFGPOD@/@COMP@.pod
@LCFGCOMP@/@COMP@
@LCFGDATA@/*
@CONFIGDIR@/
@LCFGCLIENTDEF@/@COMP@-@SCHEMA@.def
# These files are only included because we want to include the
# source files as documentation since this is an example
%doc Makefile specfile example.cin example.pod config.mk

%package defaults-s@SCHEMA@
Summary: Default resources for @NAME@
Group: @GROUP@/Defaults
Prefix: @LCFGSERVERDEF@
BuildArch: noarch
%description defaults-s@SCHEMA@
Default resources for the LCFG example component.
@LCFGCONFIGMSG@
%files defaults-s@SCHEMA@
%defattr(-,root,root)
@LCFGSERVERDEF@/@COMP@-@SCHEMA@.def

%clean
rm -rf $RPM_BUILD_ROOT
```

Appendix J

Software Package Lists

The following lists show the current versions of the packages required for an LCFG installation. The *core* packages are required to run the basic LCFG framework and are sufficient to follow the tutorial in chapter 3. The *standard* packages are the extra packages required to completely configure a basic client. The *additional* and *contributed* packages provide further optional components. G Packages marked *(P)* are pre-requisite packages which are required by some of the LCFG components in the category, but are not themselves part of the LCFG distribution. Packages marked *(S)* are component default files which need to be present (only) on the LCFG server.

J.1 Redhat 9 Packages

Core Packages (Redhat 9)

`lcfg-authorize-0.99.5-1` .... Basic Authorization module for LCFG
`lcfg-authorize-defaults-s1-0.99.5-1` *(S)* .... Default resources for lcfg-authorize
`lcfg-buildtools-2.0.34-1` ... Tools for Building and Packaging LCFG Modules from the CVS Repository
`lcfg-client-2.1.35-1` .... LCFG profile client
`lcfg-client-defaults-s2-2.1.15-1` *(S)* .... Default resources for lcfg-client
`lcfg-client-defaults-s3-2.1.35-1` *(S)* .... Default resources for lcfg-client
`lcfg-example-1.1.4-1` .... An Example LCFG component
`lcfg-example-defaults-s1-1.1.4-1` *(S)* .... Default resources for lcfg-example
`lcfg-file-1.0.9-1` .... The LCFG file component
`lcfg-file-defaults-s1-1.0.9-1` *(S)* .... Default resources for lcfg-file
`lcfg-inventory-1.1.3-1` .... LCFG inventory component
`lcfg-inventory-client-1.1.3-1` .... Client-side modules and applications for the LCFG inventory
`lcfg-inventory-defaults-s1-1.1.3-1` *(S)* .... Default resources for lcfg-inventory
`lcfg-logserver-1.1.12-1` .... LCFG logserver
`lcfg-logserver-defaults-s1-1.1.12-1` *(S)* .... Default resources for lcfg-logserver
`lcfg-ngeneric-1.1.23-1` .... LCFG new generic component
`lcfg-ngeneric-defaults-s1-1.1.23-1` *(S)* .... Default resources for lcfg-ngeneric
`lcfg-om-0.3.14-1` .... Component execution manager for LCFG
`lcfg-om-defaults-s1-0.3.14-1` *(S)* .... default resources for components which use om
`lcfg-perlex-1.1.3-1` .... An Example LCFG component in Perl
`lcfg-perlex-defaults-s1-1.1.3-1` *(S)* .... Default resources for lcfg-perlex
`lcfg-server-2.1.64-1` .... LCFG server component
`lcfg-server-defaults-s2-2.1.3-1` *(S)* .... Default resources for lcfg-server
`lcfg-server-defaults-s3-2.1.64-1` *(S)* .... Default resources for lcfg-server
`lcfg-utils-1.1.23-1` .... LCFG resources libraries and utilities
`perl-Time-modules-2003.1126-1` *(P)* .... Time-modules module for perl
`perl-W3C-SAX-XmlParser-0.99-3` *(P)* .... W3C-SAX-XmlParser module for perl
`perl-W3C-Util-Basekit-0.91-3` *(P)* .... W3C-Util-Basekit module for perl

Standard Packages (Redhat 9)

`lcfg-alias-1.0.0-1` .... The LCFG mail alias component
`lcfg-alias-defaults-s1-1.0.0-1` *(S)* .... Default resources for lcfg-alias
`lcfg-auth-0.100.8-1` .... LCFG auth component
`lcfg-auth-defaults-s1-0.100.8-1` *(S)* .... Default resources for lcfg-auth
`lcfg-boot-1.1.30-1` .... LCFG boot component
`lcfg-boot-defaults-s2-1.2.7-1` *(S)* .... Default resources for lcfg-boot
`lcfg-buildinstallroot-0.99.8-1` .... Script to rebuild the LCFG installroot
`lcfg-cron-1.1.4-1` .... LCFG cron component
`lcfg-cron-defaults-s2-1.1.4-1` *(S)* .... Default resources for lcfg-cron
`lcfg-dhclient-0.91.15-1` .... A component for configuring the dhclient object
`lcfg-dns-6.1.39-1` .... The DNS LCFG component
`lcfg-dns-defaults-s2-6.1.38-1` *(S)* .... Default resources for lcfg-dns
`lcfg-foomatic-0.99.9-1` .... The LCFG foomatic component
`lcfg-foomatic-defaults-s1-0.99.9-1` *(S)* .... Default resources for lcfg-foomatic
`lcfg-fstab-1.1.22-1` .... LCFG fstab component
`lcfg-fstab-defaults-s2-1.1.22-1` *(S)* .... Default resources for lcfg-fstab
`lcfg-gdm-0.99.20-1` .... LCFG gdm component
`lcfg-gdm-defaults-s1-0.99.20-1` *(S)* .... Default resources for lcfg-gdm
`lcfg-grub-1.2.3-1` .... component for controlling the grub bootloader
`lcfg-grub-defaults-s1-1.2.3-1` *(S)* .... Default resources for lcfg-grub
`lcfg-grub-defaults-s2-1.4.0-1` *(S)* .... Default resources for lcfg-grub
`lcfg-hackparts-0.100.8-1` .... =LCFG partition creation software
`lcfg-hardware-0.100.4-1` .... LCFG hardware component
`lcfg-hardware-defaults-s2-0.100.4-1` *(S)* .... Default resources for lcfg-hardware
`lcfg-init-0.100.2-1` .... LCFG init component
`lcfg-init-defaults-s1-1.0.4-1` *(S)* .... Default resources for lcfg-init
`lcfg-install-0.100.15-1` .... LCFG install component
`lcfg-install-defaults-s1-0.100.15-1` *(S)* .... Default resources for lcfg-install
`lcfg-kerberos-1.32.22-1` .... LCFG kerberos component
`lcfg-kerberos-defaults-s5-1.32.22-1` *(S)* .... Default resources for lcfg-kerberos
`lcfg-kernel-0.101.6-1` .... LCFG kernel component
`lcfg-kernel-defaults-s1-0.101.6-1` *(S)* .... Default resources for lcfg-kernel
`lcfg-lcfginit-0.99.4-1` .... Initialize LCFG
`lcfg-lprng-0.99.38-1` .... LCFG LPRng LPD component
`lcfg-lprng-defaults-s1-0.99.38-1` *(S)* .... Default resources for lcfg-lprng
`lcfg-mailng-1.7.3-1` .... LCFG mail component
`lcfg-mailng-defaults-s1-1.6.25-1` *(S)* .... Default resources for lcfg-mailng
`lcfg-mailng-defaults-s2-1.7.3-1` *(S)* .... Default resources for lcfg-mailng
`lcfg-network-1.99.8-1` .... LCFG network component
`lcfg-network-defaults-s2-1.99.8-1` *(S)* .... Default resources for lcfg-network
`lcfg-nsswitch-0.100.6-1` .... LCFG nsswitch component
`lcfg-nsswitch-defaults-s2-0.100.6-1` *(S)* .... Default resources for lcfg-nsswitch
`lcfg-nsu-2.4.2-1` .... nsu command
`lcfg-nsu-defaults-s1-2.4.2-1` *(S)* .... Default resources for lcfg-nsu
`lcfg-ntp-2.1.13-1` .... The NTP LCFG component
`lcfg-ntp-defaults-s2-2.1.15-1` *(S)* .... Default resources for lcfg-ntp
`lcfg-pam-1.0.5-1` .... LCFG pam component
`lcfg-pam-defaults-s2.0-1.0.5-1` *(S)* .... Default resources for lcfg-pam
`lcfg-routing-3.3.40-1` .... The routing component
`lcfg-routing-defaults-s2-3.3.39-1` *(S)* .... Default resources for lcfg-routing
`lcfg-rpmcache-1.1.17-1` .... The LCFG rpm cache component
`lcfg-rpmcache-defaults-s1-1.1.5-1` *(S)* .... Default resources for lcfg-rpmcache
`lcfg-rpmcache-defaults-s2-1.1.17-1` *(S)* .... Default resources for lcfg-rpmcache
`lcfg-sshd-1.20.4-1` .... LCFG sshd component
`lcfg-sshd-defaults-s2-1.20.4-1` *(S)* .... Default resources for lcfg-sshd
`lcfg-syslog-1.1.0-1` .... LCFG syslog component
`lcfg-syslog-defaults-s2-1.1.0-1` *(S)* .... Default resources for lcfg-syslog
`lcfg-tcpwrappers-0.99.5-1` .... LCFG tcpwrappers component

`lcfg-tcpwrappers-defaults-s1-0.99.5-1` *(S)* .... Default resources for lcfg-tcpwrappers
`lcfg-updaterpms-0.100.29-1` .... LCFG updaterpms component
`lcfg-updaterpms-defaults-s2-0.100.29-1` *(S)* .... Default resources for lcfg-updaterpms
`lcfg-xfree-1.0.0-1` .... LCFG amd component
`lcfg-xfree-defaults-s1-1.0.0-1` *(S)* .... Default resources for lcfg-xfree
`lcfg-xinetd-0.99.7-1` .... LCFG xinetd component
`lcfg-xinetd-defaults-s1-0.99.7-1` *(S)* .... Default resources for lcfg-xinetd
`netgroup-1.0-3` *(P)* .... Lists entries in NIS netgroups
`updaterpms-2.101.17-1` *(P)* .... Utilities to manage installed RPMs

Additional Packages (Redhat 9)

`Libnet-1.0.2a-2` *(P)* .... Libnet - low level network library
`PerlTk-800.024-2` *(P)* .... Perl Tk module
`lcfg-amd-0.100.10-1` .... LCFG amd component
`lcfg-amd-defaults-s2-0.100.10-1` *(S)* .... Default resources for lcfg-amd
`lcfg-apache-1.1.7-1` .... LCFG apache component
`lcfg-apache-defaults-s1-1.1.7-1` *(S)* .... Default resources for lcfg-apache
`lcfg-apm-0.100.5-1` .... LCFG apm component
`lcfg-apm-defaults-s1-0.100.5-1` *(S)* .... Default resources for lcfg-apm
`lcfg-arpwatch-1.99.13-1` .... The arpwatch component
`lcfg-arpwatch-defaults-s1-1.99.13-1` *(S)* .... Default resources for lcfg-arpwatch
`lcfg-defetc-0.100.2-1` .... LCFG default etc files
`lcfg-devlabel-defaults-s1-0.99.0-1` *(S)* .... Default resources for lcfg-devlabel
`lcfg-dialup-0.99.12-1` .... LCFG dialup component
`lcfg-dialup-defaults-s1-0.99.12-1` *(S)* .... Default resources for lcfg-dialup
`lcfg-divine-3.5.31-1` .... LCFG divine component
`lcfg-divine-defaults-s1-3.5.31-1` *(S)* .... Default resources for lcfg-divine
`lcfg-etcservices-0.99.4-1` .... LCFG etcservices component
`lcfg-etcservices-defaults-s1-0.99.4-1` *(S)* .... Default resources for lcfg-etcservices
`lcfg-ipfilter-0.0.21-1` .... The ipfilter LCFG component
`lcfg-ipfilter-defaults-s1-0.0.21-1` *(S)* .... Default resources for lcfg-ipfilter
`lcfg-iptables-0.0.77-1` .... The iptables LCFG component
`lcfg-iptables-defaults-s1-0.0.80-1` *(S)* .... Default resources for lcfg-iptables
`lcfg-irda-0.99.4-1` .... The LCFG IrDA component
`lcfg-irda-defaults-s1-0.99.4-1` *(S)* .... Default resources for lcfg-irda
`lcfg-ldap-2.0.28-1` .... LCFG ldap component
`lcfg-localhome-2.0.9-1` .... LCFG localhome component
`lcfg-localhome-defaults-s3-2.0.8-1` *(S)* .... Default resources for lcfg-localhome
`lcfg-nfs-1.0.2-1` .... LCFG nfs component
`lcfg-nfs-defaults-s2-1.0.1-1` *(S)* .... Default resources for lcfg-nfs
`lcfg-nscd-1.5.5-1` .... LCFG nscd component
`lcfg-nscd-defaults-s2-1.5.5-1` *(S)* .... Default resources for lcfg-nscd
`lcfg-nut-defaults-s2-2.1.55-1` *(S)* .... Default resources for lcfg-nut
`lcfg-pcmcia-0.100.2-1` .... LCFG pcmcia component
`lcfg-pcmcia-defaults-s1-0.100.2-1` *(S)* .... Default resources for lcfg-pcmcia
`lcfg-ramdisk-1.3.0-1` .... LCFG ramdisk component
`lcfg-ramdisk-defaults-s1-1.3.0-1` *(S)* .... Default resources for lcfg-ramdisk
`lcfg-rmirror-1.8.9-1` .... This is the LCFG component for the rmirror service.
`lcfg-rmirror-defaults-s2-1.8.6-1` *(S)* .... Default resources for lcfg-rmirror
`lcfg-rpmaccel-0.99.3-1` .... LCFG rpmaccel component
`lcfg-rpmaccel-defaults-s1-0.99.3-1` *(S)* .... Default resources for lcfg-rpmaccel
`lcfg-rsync-2.1.0-1` .... LCFG rsync component
`lcfg-rsync-defaults-s2-2.1.0-1` *(S)* .... Default resources for lcfg-rsync
`lcfg-schemes-0.99.18-1` .... LCFG network scheme handling
`lcfg-snmp-3.1.4-1` .... The snmp LCFG component
`lcfg-snmp-defaults-s2-3.1.4-1` *(S)* .... Default resources for lcfg-snmp
`lcfg-symlink-0.100.6-1` .... LCFG symlink component
`lcfg-symlink-defaults-s2-0.100.6-1` *(S)* .... Default resources for lcfg-symlink

`lcfg-vlan-0.100.0-1` .... LCFG vlan component
`lcfg-vlan-defaults-s1-0.100.0-1` *(S)* .... Default resources for lcfg-vlan
`lcfg-ypclient-0.99.7-1` .... LCFG ypclient component
`lcfg-ypclient-defaults-s1-0.99.7-1` *(S)* .... Default resources for lcfg-ypclient
`perl-Expect-1.15-1` *(P)* .... Expect module for perl
`perl-IO-Tty-1.02-1` *(P)* .... IO-Tty module for perl
`pump-0.8.11-1` *(P)* .... A Bootp and DHCP client for automatic IP configuration.

Contributed Packages (Redhat 9)

`lcfg-bluez-0.99.11-1` .... LCFG mail component
`lcfg-bluez-defaults-s1-0.99.11-1` *(S)* .... Default resources for lcfg-bluez
`lcfg-toshset-0.99.3-1` .... The LCFG toshset component
`lcfg-toshset-defaults-s1-0.99.3-1` *(S)* .... Default resources for lcfg-toshset
`lcfg-vigor-0.99.12-1` .... The LCFG Vigor component
`lcfg-vigor-defaults-s1-0.99.12-1` *(S)* .... Default resources for lcfg-vigor
`lcfg-vmidi-0.99.6-1` .... The LCFG VMIDI component
`lcfg-vmidi-defaults-s1-0.99.6-1` *(S)* .... Default resources for lcfg-vmidi

Bibliography

[And94] Paul Anderson. Towards a high-level machine configuration system. In *Proceedings of the 8th Large Installations Systems Administration (LISA) Conference*, pages 19–26, Berkeley, CA, 1994. Usenix.
http://www.lcfg.org/doc/LISA8_Paper.pdf.

[And00] Paul Anderson. Large scale Linux configuration management. *Linux Journal*, pages 58–62, May 2000.
http://interactive.linuxjournal.com/Magazines/LJ72/3467.html.

[And01] Paul Anderson. Dice and lcfg software guidelines. Internal Document, 2001.
http://www.dice.informatics.ed.ac.uk/doc/dice-guidelines.pdf.

[AS00] Paul Anderson and Alastair Scobie. Large scale Linux configuration with LCFG. In *Proceedings of the Atlanta Linux Showcase*, pages 363–372, Berkeley, CA, 2000. Usenix.
http://www.lcfg.org/doc/ALS2000.pdf.

[AS02] Paul Anderson and Alastair Scobie. LCFG - the Next Generation. In *UKUUG Winter Conference*. UKUUG, 2002.
http://www.lcfg.org/doc/ukuug2002.pdf.

[Har03] Angus W Hardie. LCFG for Mac OS X. Undergraduate project report, June 2003.
http://www.lcfg.org/doc/hardie.pdf.

[Mic] Sun Microsystems. *Solaris 9 Installation Guide*. Sun Microsystems.
http://docs.sun.com/db/doc/816-7171/6md6pohq.

Index