LCFG: The Next Generation

by Paul Anderson <paul.anderson@ed.ac.uk>
Alastair Scobie <ajs@dcs.ed.ac.uk>

Division of Informatics
University of Edinburgh

1 Abstract

This paper describes the most recent version of the LCFG automatic configuration and installation system, originally developed at Edinburgh University around 1993. This version contains a number of significant differences from the original, and has become known as LCFGng (next generation).

LCFG provides a configuration language and a central repository of configuration specifications, from which individual Unix machines can be automatically installed and configured. Changes to the central specification automatically trigger corresponding changes in the actual configuration of individual nodes. The system is particularly suitable for sites where the configurations are very diverse (ranging from large servers to laptops), and different aspects of the configurations may be changed frequently, and managed by many different people. LCFG scales to at least medium-size sites (~1000 nodes).

The paper first presents some background and history of the LCFG framework, followed by a general discussion of the large-scale configuration problem. The current implementation of LCFG is then described, noting where, and why, this differs from the original version. Finally, current and future development plans are discussed.

2 Background

For many years, the Unix community has recognised the inadequacy of vendor-supplied configuration tools for managing large networks of disparate machines. A wide range of solutions have been proposed and developed by systems administrators, frequently just for their own use. Past proceedings of the Usenix LISA conference[1] give some idea of the scope of these tools, which range from simple cloning mechanisms to more complex database systems.

The LCFG framework [And94] was developed by The Department of Computer Science at Edinburgh University to handle their own network of several hundred (mostly Solaris) Unix machines. This system was designed to satisfy several fundamental properties that we could not find in any existing implementation, and has been very successful. As the Edinburgh site migrated towards Linux, the LCFG framework was ported and extended [AS00], so that Linux is now the primary platform and the Solaris version is no longer supported[2]. Recent expansion in the local use of LCFG has prompted a major overhaul known as LCFGng. This includes a move to XML/HTTP for transporting configuration profiles, and the rewriting of much of the basic framework. This paper updates [AS00] to include these changes.

LCFG is now being used more widely outside of Edinburgh, including its adoption as the configuration system for the testbeds of the European DataGRID.[3] The latest code and documentation is designed to be suitable for public use (under the GPL) and an LCFG web site at `http://www.lcfg.org` is under development for its distribution.

3 System Configuration

When a standard release of a large software product, such as an operating system, is distributed to many users, it invariably needs *configuring* to tailor it to the requirements of each individual installation. Even within a single site, there can be a huge difference between configurations of different machines; the following are some examples of the parameters which may vary:

- Hardware configuration and drivers.
- Network configuration and servers.
- Installed software.
- Network services provided.
- Access control.

[1] http://www.usenix.org/events/bytopic/lisa.html

[2] We would be interested in resurrecting Solaris support, but we currently do not have the necessary effort

[3] http://www.datagrid.cnr.it/

3.1 Manual Configuration

Obviously the amount of variety between machines depends on the type of installation; an academic Computer Science environment tends to have a larger variety of software and configurations than a site concerned primarily with a single application. Our current LCFG system supports over 2000 parameters, about 25% of which routinely vary between different systems. For individual machines, or a small site, these parameters would traditionally be configured manually, and most distributions include graphical tools to make this process more straightforward (for example [Cal, Sol]). However, large installations require some degree of automation.

3.2 Automatic Configuration

Most obviously, the effort required to configure several hundred machines manually from a graphical interface is not normally acceptable. Most sites will also want to have a reasonable confidence in the correctness of their configurations; misconfigured systems represent serious security problems, as well as leading to unpredictable failures. Manually configured systems, with no explicit representation of the configuration, are notoriously difficult to guarantee correct. Early attempts to overcome these problems were often based on a *cloning* procedure where a single machine is configured by hand and the resulting disk image is copied directly onto a set of other machines. This is usually followed by execution of some scripts to apply any machine-specific differences. It can be argued that this process is useful for large numbers of very similar machines which do not change regularly, but we believe that configuration changes, and differences, are sufficiently common in most practical cases to override any advantages of a pure cloning mechanism.

Our own installation is one example where both the variety of different configurations, and the rate at which they change, makes cloning impractical. We support a range of machines from file servers, to student laboratory clients, to researcher's laptops, whose hardware and software requirements are all very different. New machines arrive continuously, old machines are re-allocated, and systems are rebuilt after hardware failures or OS upgrades; all of these imply a reconfiguration, and we estimate that, on average, about 10% of our machines are completely reconfigured each week. Small configuration changes also occur very frequently in a complex environment; for example, changing a server or gateway can imply configuration changes for many other hosts. Software updates also occur at an average rate of several tens of packages per day.

3.3 Supporting Diversity & Change

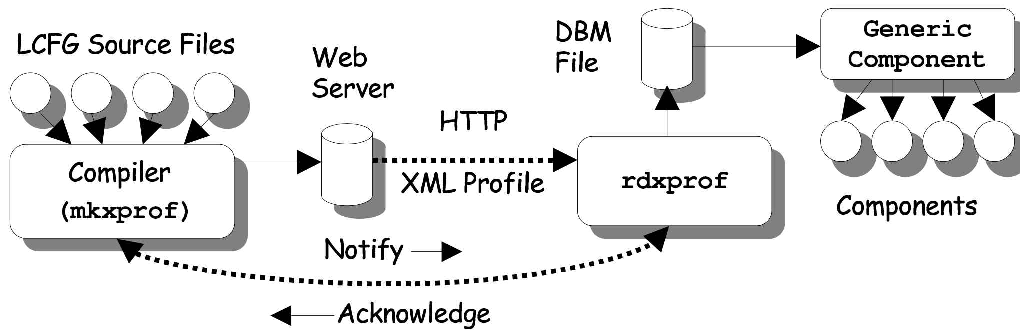
Automatically supporting such diversity and rate of change requires two main features from a configuration system: Firstly, there must be some representation of the configuration information which is stored independently of the host systems. This may be simple copies of machine configuration files stored on a central server, or it may be a more complex "database". Secondly, the system must be capable of tracking changes to the configuration and applying them to individual machines as necessary, rather than requiring an explicit reconfiguration operation.

An explicit, declarative representation of the configuration information is essential, and the format and the structure of this are very important; An obvious solution is to store configuration files in the same format as they appear on the target host, but this coarse grain approach means that the configuration system is not aware of the relationships between data in the various files; for example, the "owner" of a machine may have special access rights which involves the username appearing in several different configuration files, and we would like to be able to change this at a single point. Storing the configuration specifications in a more abstract format allows the configuration to be examined and manipulated in a more meaningful way. It also provides a degree of platform independence, analogous to using a high-level programming language which can be transformed into code for any specific platform. The configuration system may use this information to generate traditional configuration files or the services on the host machine may be modified to read this configuration format directly.

A class structure is usually the most convenient way of organising the configuration information. Hosts usually fall into various different categories (laptop, web server, student desktop, etc.) and it is natural to specify new machines by using inheritance to describe just the difference (if any) between the new machine and some existing category. Many systems adopt this approach with varying degrees of sophistication (for example [Mic97, Bur95]). Once the information is available in this high-level form, there is considerable potential for analysing and generating the specification automatically. This opens up the possibility of treating the configuration of a whole site as a complete entity; for example, we should be able to prevent a gateway being removed while there are still clients which depend on it.

The process of actually changing a machine configuration to match a particular specification is not normally straightforward. Ideally, we would like this to take place automatically as soon as the specification

Figure 1: LCFG Architecture



is changed. Sometimes this is possible; for example changing an entry in a TCP wrapper. However, changing the disk partitioning is probably not desirable, or even possible, while a machine is in use. In practice, changes take place at different times, as appropriate; sometimes immediately, sometimes from a nightly cron job and sometimes at reboot. Laptops are an interesting case because they can normally only be reconfigured while they are connected to the network, and this might not happen very often at boot time, or at the time when a nightly cron job would normally run. We allow laptops to be reconfigured on demand by the user so that updates take place at a convenient time.

4 LCFG

LCFG is designed to handle automated installation and configuration in a very diverse and evolving environment. Figure 1 shows the overall architecture of the LCFG framework:

- The configuration of the entire site is described in the source files, held on a central server. Note that one source file does not (necessarily) correspond to one machine, or one component; a source file typically describes one *aspect* of the overall configuration, that usually forms a logical management unit, such as “parameters specific to student machines”, or “parameters specific to Redhat 6.2 machines”. An individual parameter may be affected by more than one one source file.

- The source files are compiled into individual *profiles*. One profile corresponds to one machine, and the profile contains all the configuration parameters necessary to recreate the configuration of the target machine. The profiles are published on a web server.

- When a profile changes, the corresponding client is sent a simple UDP notification. The client retrieves the profile using HTTP, and caches the parameters in a DBM file. Clients normally poll periodically for new configurations in case the notification has been missed.

- Clients periodically send a simple UDP acknowledgement to the server. These are collated to generate a web page showing status information for all the clients.

- *Component* scripts on the client are responsible for reading the configuration parameters and taking the appropriate actions necessary to implement the configuration; usually this involves generating configuration files from the parameters in the profile. Components are notified when a new configuration is received which involves a change to some parameter of that component. The component regenerates any necessary configuration files, and notifies any associated daemons.

4.1 The Profile

The LCFG profile is an XML document with a simple structure, containing one section corresponding to each component. The configuration of each component is described by a list of key-value pairs, with the keys and default values being defined by the component author. The profile provides an explicit, declarative definition of the entire machine configuration (it is not intended to be written or read by humans, but it can be if required).

The current LCFG profile schema [Col01] is a subset of an experimental schema defined in conjunction with the European DataGRID. This has the advantage that it is possible to experiment with new configuration languages and to test the results by compiling them into the common profile format. It is also possible to experiment with new client-end implementations, by using

the existing compiler.

Earlier versions of LCFG used an NIS map to transfer the resources. Not only did this require NIS on the server and all the clients, but since the map contained values for *all* clients, this solution was not scalable.

4.2 The Source Files

Research is currently underway to design a new specification language (see 5.1) for LCFG (the use of the standard profile makes it easy to use alternative languages). In the meantime, the current version uses a sightly updated form of the original LCFG specification language:

The configuration parameters are specified by key-value pairs known as *resources*. These are inspired by X resources, and similar to the parameters used by COAS. For example:

```
mambo.dns.servers localhost
```

The key specifies the hostname, the LCFG *component*, and the parameter. The configuration files are passed through the C preprocessor, supporting simple inheritance by file inclusion:

```
#include <standard_laptop.h>
auth.owner paul
....
```

The machine-specific file need only list those resources which are different from a standard laptop (the first component of the resource keys is generated automatically from the name of the file). Notice that the included header file relates to a high-level *aspect* (standard laptop machine, in this case) which may contain resource specifications for many different low-level components. The header files may of course be nested.

Together with the use of preprocessor variables, this simple mechanism provides a powerful way of presenting complex host configurations in a clear way, with very little specialised software. However, it is not sufficiently fine-grained to process information inside the resource keys. This causes difficulties in some cases; for example, a standard configuration might specify that the CD-ROM should have its ownership changed to match that of the user at the console:

```
auth.consolepermclass_cdrom
  /dev/cdrom
```

If we have a second SCSI CD on our machine then we might want to specify:

```
auth.consolepermclass_cdrom
  /dev/cdrom /dev/scd0
```

Specifying this directly for a particular host is not good, because it overrides the specification for the standard machine, so that changes to the standard specification (such as changing the location of the default CD-ROM) would not be reflected on this particular host.

Resources values can be manipulated by specifying a Perl regular expression which is applied to any previously declared value to change it is some way. This allows us to easily append or prepend items to a standard value:

```
auth.consolepermclass_cdrom
  !/(.*)/\$1 /dev/scd0
```

We call this process *mutation*. Although it is very powerful, overuse can lead to configurations which are very hard to understand, and we usually restrict its use to a few well-defined macros:

```
auth.consolepermclass_cdrom
  ANDALSO(/dev/scd0)
```

The LCFG source files for the entire installation are maintained on a central server under RCS control. The compiler monitors the source files for changes at regular intervals, and uses a cache of dependency information to determine which profile may be affected by any change, and hence require recompilation (See [And01b]).

4.3 LCFG Components

Each subsystem on a host (for example, the mail system), has a controlling *component script* which performs some action on the subsystem, according to the *method* supplied as an argument. This is very similar the System V init mechanism, and the standard methods include `start` and `stop`, for example, which are called at appropriate times to allow the component to start and stop associated daemons. The `configure` method of a component is called when the resources change, and the component should read the resources from the profile and configure the appropriate subsystem. This may involve translating the configuration parameters into a traditional configuration file, or controlling a service directly; for example, restarting some daemon with command-line parameters derived from the configuration resources.

Shell script is normally the most suitable language for LCFG components, although the new implementation

of the framework is based on Perl modules which are used to construct the shell commands used by the components. This means that the supporting infrastructure is available to native Perl components and we expect to have official support for pure Perl components very soon.

Shell components are constructed by including code for a *generic* component which implements default functions for all the standard methods and performs common operations such as loading profile resources into environment variables. Components need only redefine the necessary methods. A special macroprocessor (`sxprof`) is provided which can take a template configuration file and substitute values directly from the profile resources. Many components can be implemented simply by calling this template processor to generate the appropriate configuration files, in response to the `configure` method. Appendix B shows a simple component example. The `start` and `stop` methods would not be required unless the component was managing a daemon.

Most scripts are quite short and it is easy to write a configuration script for a new package or service. Initially, new scripts are normally written just to support the subset of configuration parameters which are expected to vary at the local site, and this is easily expanded later as the need arises. The independence, and ease with which these scripts can be created has been a major reason for the success of the system; they are usually created by the person responsible for the corresponding service, and many people have contributed; we currently have around 50-60 active components.

4.4 Software Updates

One component is responsible for maintaining the software packages on the client according to the central specification; this component is based on the locally-developed `updaterpms` program (currently written in C and using `rpmlib` [4]). This update process was developed for the Linux port of LCFG when it replaced the `lfu` [And91] system used in the original Solaris version.

`Updaterpms` compares the RPMs installed on a machine with a specification provided by the LCFG and installs, upgrades, or deletes RPMs as necessary. In the current implementation, the RPM list is contained in a separate file which is specified by a resource in the profile; the complete list of RPMs is not stored in the profile itself, although this is the intention in the future. The client machine also requires access to a filesystem (usually remote) containing both the RPMs themselves, and the file specifying the list of RPMs which should be installed.

The package specification files are preprocessed with the C preprocessor to provide some degree of structure similar to the LCFG source files themselves. Individual machines can therefore include a standard software specification and override individual packages. The RPM specifications may contain wildcards[5] to refer to the latest version (or release); for example, the standard installation might include a specific version:

```
toshutils-1-1.34
```

And a particular machine might override that to carry the latest available version:

```
#include <standard>
+toshutils-1-*
```

The '+' symbol indicates that the new specification overrides any preceeding one thus inhibiting the error message that would normally be generated by the duplicate package specifications.

The ability to import software which has been prepackaged in RPM format saves a considerable amount of work. However the packaging is not always well implemented; post-install scripts, for example, are often poorly designed, perhaps attempting to add users to a password file, or to demand user interaction, neither of which are appropriate for an automated install on a networked system. Occasionally, dependency information is also incorrect. Several standard RPM options such as `--noscripts` or `--force` can be specified on a per-package basis to help with these problems.

4.5 System Installation

Providing the ability to install new machines with the absolute minimum of manual intervention is very important. This allows failed machines to be replaced, and new machines to be installed, quickly and correctly, by unskilled staff [6].

Installing an operating system on to bare hardware requires some sort of bootstrap process. Typically:

- ❑ A minimal version of the operating system or other install program is loaded from the network, or removable media.

[4] The latest version of `updaterpms` needs a modified version of `rpmlib` to support per-package options

[5] This is one reason why the replacement of the remote filesystem protocol with, for example HTTP, is not completely straightforward.

[6] This does not include the restoration of any user data from backup

- Once booted, this program partitions the system disk and installs a copy of the operating system.

- There will usually be some additional software installation and configuration, the first time that the machine reboots from the newly installed system.

LCFG can use floppy disk, or CD ROM to boot a bare machine, using an NFS-mounted root partition (the latest version of LCFG can also use PXE). When the minimal system has booted, an `update` component runs automatically to partition the system disk according to the LCFG resources and `updaterpms` loads the software. In the original Solaris implementation, a small hand-crafted image was first copied directly from the network onto the system disk, but this is difficult to maintain. In the current version, the `updaterpm` component builds the root filesystem completely from a set of RPMs. Although it is aware of the context, this uses exactly the same process which is used nightly to update the software on a running machine, ensuring a consistent interpretation of the LCFG resources at install and update time.

When the software has been installed, the system reboots from the new image. The LCFG components start normally and perform the remaining configuration. Once the install operation has been started, it runs completely unattended, allowing whole laboratories of machines to be installed easily by one person, even if the machines have very different configurations.

This installation process was developed for the Linux port of LCFG since the installation process is very OS-specific and the original Jumpstart [Mic97] based procedure used under Solaris was not suitable. Note that Kickstart [Ham] is not used.

5 Some Issues and Future Developments

As well as providing a practical tool for management of large site configurations, LCFG acts as a testbed for ongoing research and development in system configuration. A recent workshop on Large Scale System Configuration [AP01] covered many of these active issues:

5.1 Languages

The LCFG profile provides a convenient common "output language" for compilers of high-level specification languages. This is an area of active research and some recent work includes [And01a], and the proposed DataGRID language [CP02].

5.2 Scheduling and Monitoring Changes

One major issue with the current LCFG implementation is the difficulty of deciding when a node should be permitted to implement a configuration change, as described in section 3.3. It is also not clear whether a node should be allowed to implement parts of configuration change, or should defer all changes until they can be implemented as one transaction. These issues are being explored in collaboration with the DataGRID.

A closely related issue is that of monitoring the state of node configurations. LCFG provides facilities to monitor when a node has received a new profile, but there is no easy way of determining whether all components on the node have actually implemented this profile.

5.3 Configuration Queries

The profile provides a clear and simple definition of the complete configuration for a single node, but it not suitable for making queries which span across all nodes; for example, "which machines are using server X as their primary DNS server"? We would like to provide views of the configuration data in a form that is more suitable for this type of operation, such as LDAP.

5.4 Software Updates

We would like to include the RPM lists in the profile and to provide a transport protocol which does not depend on the existence of a remote filesystem for retrieving RPMs.

5.5 Dynamic Configuration Data and Contexts

Most configuration parameters are most appropriately stored in the central specification. However, certain parameters may be *dynamic* in that they are generated on the client, and it is not appropriate to refer to a master copy on the configuration server. For example, a disconnected laptop might reconnect via a different ISP and need to use configuration parameters supplied by the ISPs DHCP server instead of the default values used on the home network. Service location protocols may also be an important technique for reconfiguring machines in very large clusters in the event of a service failure; in this case, some configuration parameters would be obtained by the client directly via the SLP, rather than the configuration server.

In some cases, a node may also need to adopt one of several alternative configurations at different times. For example, a laptop may switch between an Ether-

net connection, and a dialup connection. A Grid node may switch between different system software configurations, depending on the job cluster to which the node is allocated. It is therefore useful to support the idea of *contexts* which allow the central specification to define several variations for an individual node configuration. The selection between these variations can then be made by changing the context on the node at runtime without changing the central database.

The current version of LCFG includes experimental support for contexts and dynamic parameters, but further work is required to identify the best way to implement these.

Appendix B An Example Component

```
#!/bin/sh
########################################################################
#
# Example LCFG Component
#
# Paul Anderson <paul@dcs.ed.ac.uk>
# Version 1.0.1 : 11/01/02 15:58
#
#  ** Generated file : do not edit **
#
########################################################################

. /etc/obj/ngeneric

###########################################################################
Configure() {
###########################################################################

  # Make directory for config file if it doesn't exist
  mkdir -p /var/obj/conf/example >>$_LOGFILE  2>&1
  [ $? = 0 ] || Fail "failed to create config directory (see logfile)"

  # Use sxprof to substitute the configuration parameters from the
  # environment into the template.
  /usr/bin/sxprof -i $_COMP /usr/lib/lcfg/conf/example/template \
                             /var/obj/conf/example/config >>$_LOGFILE 2>&1

  # 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
  return;
}

###########################################################################
Run() {
###########################################################################

  # Stop daemon here
  return;
}

###########################################################################
# Dispatch methods
###########################################################################

Dispatch "$@"
```

Appendix B References

[And91] Paul Anderson. Managing program binaries in a heterogeneous Unix network. In *Proceedings of the 5th Large Installations Systems Administration (LISA) Conference*, pages 1–9, Berkeley, CA, 1991. Usenix Association.
http://www.dcs.ed.ac.uk/~paul/publications/LISA5_Paper.pdf.

[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.

[And01a] Paul Anderson. A declarative approach to the specification of large-scale system configurations. Discussion Document, February 2001.
http://www.dcs.ed.ac.uk/~paul/publications/conflang.pdf.

[And01b] Paul Anderson. LCFG adaptors for XML profiles. LCFG Documentation, 2001.
http://www.lcfg.org/doc/lcfg-profile.pdf.

[AP01] Paul Anderson and Jessie Paterson. Large scale configuration workshop. Web page, November 2001.
http://www.dcs.ed.ac.uk/home/paul/wshop/.

[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.

[Bur95] Mark Burgess. Cfengine: a site configuration engine. *USENIX Computing systems*, 8(3), 1995.
http://www.iu.hioslo.no/~mark/research/cfarticle/cfarticle.html.

[Cal] Caldera. COAS. Web page.
http://www.coas.org.

[Col01] Tim Colles. Node profile specification. European DataGRID WP4, Internal Paper, July 2001.
http://www.lcfg.org/doc/np.pdf.

[CP02] Lionel Cons and Piotr Poznanski. A high level configuration description language. European DataGRID WP4, Internal Paper, January 2002.
http://hep-proj-grid-fabric-config.web.cern.ch/ ...
hep-proj-grid-fabric-config/documents/cfglan.pdf.

[Ham] Martin Hamilton. RedHat Linux Kickstart HOWTO. Web page.
http://metalab.unc.edu/pub/Linux/docs/HOWTO/other-formats/ ...
html_single/KickStart-HOWTO.html.

[Mic97] Sun Microsystems. Preparing custom Jumpstart installations. In *Solaris 2.6 Advanced Installation Guide*, pages 71–126. Sun Microsystems, 1997.

[Sol] Solucorp. Linuxconf. Web page.
http://www.solucorp.qc.ca/linuxconf/.