[sage]

ISBN 1-931971-42-0

14 Paul Anderson System Configuration

14 | *Short Topics in* System Administration

Rik Farrow, Series Editor

System Configuration

Paul Anderson

[sage]

Booklets in the Series

14 *Short Topics in*
System Administration

Rik Farrow, Series Editor

System Configuration

Paul Anderson

Published by the USENIX Association
2006

ISBN 1-931971-42-0

First Printing 2006

Contents

List of Figures

Acknowledgments

Thanks to Adam Moskowitz for persuading me that it was worthwhile writing down my thoughts on this subject, and to Rik Farrow for his supportive editing. The computing staff in the School of Informatics at Edinburgh University have suffered my experimental code on their production system and provided invaluable practical feedback. Too many people to name have contributed by reviewing text, commenting, and patiently explaining things that I didn't understand.

Introduction

The complexity of a typical computing site has increased immensely over the past 10 years, not only in scale but also in the range of services and the intricacy of its interconnections. This has led to systems which can no longer be configured reliably using a purely manual process. For example:

- If a Web server fails, what needs to be done to enable a replacement? This is likely to require a complex chain of reconfigurations, involving firewalls, DNS servers, database servers, and backups. This example is discussed in more detail in section 1.4.
- Can we be sure that the appropriate configuration files on every machine are always set in such a way as to implement the desired security policy? This policy does not just involve individual machines, but also the trust relationships between them.
- Can we be sure that different people, responsible for different aspects of a site, will not change configurations in conflicting ways?
- Can we be sure that a complex service is not configured in such a way that there is a "single point of failure" which has not been identified?

In addition to this increase in complexity, new requirements are presenting new challenges:

- Higher expectations of service reliability call for systems that are able to reconfigure, without manual intervention, to cope with failures of individual machines or components (*autonomics*).
- More complex relationships between machines on an individual site, and between remote sites, imply a more collaborative approach to the development of overall configurations. These are no longer under the complete control of a single individual or small group. This leads to complex problems of security and conflict resolution (*federation*).

Without automated solutions to problems such as those listed above, large computing sites will become increasingly unreliable and unmanageable. Solving these problems is the domain of system configuration.

The State of the Art

An awareness of the need for automated system configuration has been gradually growing within the system administration community itself, and this has spawned a great deal of discussion and a plethora of "homegrown" tools. Many of these can be found in the proceedings of the LISA conferences [9]. However, the vast majority of these tools address only the lowest levels of the problem, and system configuration continues to be a major source of failures (see [60]) and to consume a disproportionate amount of highly skilled manual effort.

Senior system administrators and consultants looking for technologies and approaches to site configuration are faced with a difficult task; there is no clear process for even identifying and evaluating a potential selection of tools, and there will rarely be a single obvious candidate. Most sites end up with a number of imported tools and a large amount of locally developed "glue." Some sites will decide to develop significant tools of their own, but it is easy to underestimate the resources that this requires, in terms both of skills and experience and of the sheer effort of maintaining critical code which has close dependencies on many rapidly evolving applications: LCFG (see section 5.2), for example, has probably consumed about 10 person-years of direct development effort, supported by twice that effort in configuration-related research. Significant ongoing effort is required to track new operating system and application releases.

So far, tool development appears to have failed to attract a coordinated implementation effort from the open source community. It is also hampered by a lack of appropriate standards, which prevents interoperability and effective tool-sharing. Vendor-supplied tools are frequently used for specific tasks, but they, too, fail to address the overall problem adequately. There is no single, obvious reason for this lack of progress; certainly, the issue is a large one whose solution is well beyond the resources of a system administrator developing code part-time. However, the subject also includes many hard problems whose solution probably involves attracting more interest from computer scientists and theoreticians.

Tools designed to address the real problems of system configuration also face an acceptance challenge from the system administration community; such tools are likely to hide the low-level details of the familiar configuration files and to work with higher-level concepts. This represents a paradigm shift for the average system administrator, comparable to the assembler code programmer who must learn to write distributed applications in Java without worrying about which registers hold which variables! More predictable and trustworthy tools, with clear semantics, are necessary to earn this acceptance.

About This Booklet

This is not a "cookbook"; it does not include detailed descriptions of using specific tools to solve configuration problems, and there is no attempt to provide a comprehensive tool survey, since tools can change rapidly (for some recent evaluations see, e.g., [29], [21]). But neither is this booklet about the developing theory of system configuration (although such theories are summarized in chapter 6). Rather, the booklet is aimed squarely at the working system administrator; it aims to explain clearly the various facets of the system configuration problem and to describe how these relate to current tools and future research.

Understanding underlying configuration principles will help system administrators to use “best practice” in applying current configuration tools and procedures; these are often very flexible, and it is all too easy to negate the advantages presented by a perfectly good tool. Descriptions of the various facets of the configuration problem should also help provide system administrators with criteria to evaluate potential tools, since most sysadmins will be able to relate these to concrete examples at their own sites.

In the longer term, a better understanding of the fundamental principles should encourage theorists to address underlying configuration problems and tool developers to incorporate this theory into their products. Ultimately, this will yield new tools and enable system administrators to have more confidence in the ability of these tools to automatically manage the configuration of their sites.

Finally, it should be noted that there is not yet consensus on all aspects of the subject, and some topics may reflect the author’s personal bias. For instance, the LCFG tool is often used in the examples that follow, because it is broad enough to illustrate many of the important issues in a consistent way. However, it is hoped that the presentation is sufficiently logical and objective to allow readers to draw their own conclusions.

The LISA configuration workshops and the lssconf mailing list[1] provide some forums for the topics discussed in this book.

A Word About Operating Systems

This booklet deals mostly with principles that apply equally to any operating system. However, the terminology and most examples are taken from UNIX, and it is expected that this will be the natural background of most readers. UNIX probably has the widest range of configuration tools and practices; at the bottom end, raw UNIX systems provide no clear standard and no comprehensive support for system configuration—many sites without the necessary in-house skills still have very primitive configuration management. At the opposite extreme, the most complex sites and the most sophisticated configuration tools are probably UNIX-based.

Operating systems such as Microsoft Windows have all the same problems and requirements for configuration management, but there is less variety in the tools and approaches; vendor-supplied tools obviously tend to predominate, and these are certainly comparable to the middle ground of the UNIX range. The standard Microsoft tools, for example, are mentioned where appropriate.

The Structure of the Book

Chapter 1 is perhaps the most important chapter of this book; it defines the system configuration problem and describes its various aspects in some detail, without consid-

1. http://homepages.inf.ed.ac.uk/group/lssconf/.

ering specific solutions. This section is recommended as a prerequisite for understanding the material in the following sections.

Chapter 2 outlines the range of approaches to automating configuration solutions, starting with a largely manual approach and moving towards fully autonomic site management.

Chapter 3 describes some of the ways in which various tools approach different aspects of the configuration problem.

Chapter 4 covers tools and techniques which are specifically used for distributing files and managing software packages. Although this is only one (small) aspect of the overall configuration problem, it is very common to see such tools extended and used to provide primitive configuration management.

Chapter 5 describes a number of tools in more detail. These are chosen to demonstrate the range of different approaches rather than to provide a basis for selecting or using a particular tool.

Chapter 6 briefly describes some of the current issues in configuration theory and research. Less theoretically inclined readers may want to skip this section.

Most chapters end with a list of key points which summarize the important issues. The collection of these key points should provide a good overview of the subject.

Terms in italics are explained in the Glossary.

1. What Is System Configuration?

The term "configuration" has been used in many contexts, with many different meanings, and this has often caused some confusion. Unfortunately, there is no widely accepted name for the task which we will call "system configuration,"[1] although the fundamental problem is quite easy to describe. This chapter sets out a basic definition and contrasts it with some other uses of the term. It then goes on to discuss some of the many factors that complicate the configuration problem in practice.

1.1 The Configuration Problem

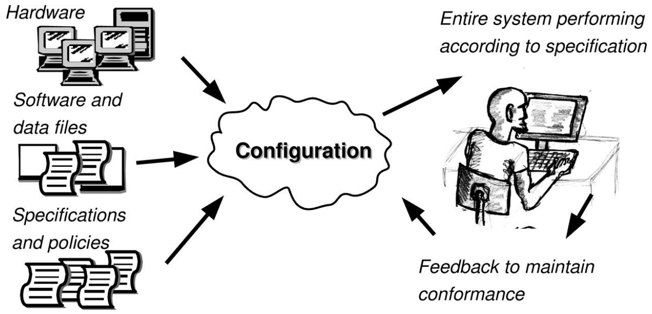

Hardware
Software and data files
Specifications and policies
Configuration
Entire system performing according to specification
Feedback to maintain conformance

Figure 1.1: The basic system configuration task

The basic system configuration problem is quite simple to describe:

- Start with:
  - A large number of varied machines with empty disks;
  - A repository of all the necessary software packages and data files;
  - A specification of the functions that the entire system is intended to perform.
- Load the software and configure the machines to provide the required functionality. Providing this externally visible functionality usually involves a good deal of internal infrastructure: for example, DNS, LDAP, DHCP, NFS and NIS services.
- Reconfigure the machines whenever the required service specification changes.

1. This seems like a good choice, because it suggests the connection with "system administration."

- Reconfigure the machines to maintain conformance with the specification whenever the environment changes—for example, when things break.

1.1.1 Other Definitions of Configuration

The term "configuration" has been used in various disciplines apart from system administration. In most cases, the usage implies a somewhat different problem from the "system configuration" described above. However, there are some similarities, and it is worthwhile mentioning a number of examples.

Hardware configuration involves selecting and arranging hardware parts to construct an overall system to a particular specification. This is subtly different from the system configuration problem, since the hardware parts tend to have fixed characteristics, whereas computer systems are "soft" and their roles can be changed by setting the appropriate parameters. Typical early work on hardware configuration includes John McDermott's expert system for configuring VAX hardware [56]. More recently, constraint programming (CSP) has been used [43]. This field has not been very active recently, but the AI and CSP techniques were precursors to some of the current system configuration research on deducing configurations to meet specific constraints (see section 6.4).

Software configuration management is also concerned with assembling complete systems (in this case, software applications) from component parts (modules). This does include the problem of ensuring that the module interfaces are compatible, but the focus tends to be more on change management. Although this is also important for system configuration, this work does not appear to have a particularly strong connection.

Network configuration management is concerned with the configuration of network devices such as switches and routers. This shares many problems with system configuration, since modern devices are highly configurable and their interaction must be managed to achieve some desired overall effect. It is possible to think of this as a subproblem of system configuration, with the additional complication that the devices are often specialized and have dedicated management protocols.

Distributed application configuration involves the deployment and configuration of processes onto distributed computer nodes to create a single distributed application. This field is highly relevant to system configuration and shares many of the same problems. In particular, the recent *CDDLM* Global Grid Forum work on configuration of *Grid* applications [5] includes language work which is directly appropriate (see section 5.4). However, the deployment model tends to assume the existence of a working infrastructure onto which the applications can be "deployed" and "undeployed." This is not a good match for configuration of the underlying infrastructure itself; it is not useful simply to "undeploy" the DNS service, for example—service configurations must be capable of evolving continuously without interrupting the service (see [26] for a discussion of these issues).

Per-user configuration refers to the setting of application preferences for individual users, as opposed to the system configuration, which is determined on a per-machine basis, usually by a machine's administrator. In some cases, such as the Microsoft registry, per-user configuration is handled in a very similar way to the per-system configuration. Other per-user tools such as GConf [4] also have aspects which are relevant to system configuration: XML formats for storing configuration data, for example, and mechanisms for notifying applications of configuration changes. In general, however, per-user configuration tends to be treated as a separate problem.

1.2 Files and Configuration Specifications

The distinction between the configuration specifications and the software packages and data files in the definition in section 1.1 is important and worth clarifying. Consider two extreme cases:

- If one wanted to control every aspect of every machine in minute detail, it would be possible to use the configuration specification to define the value of every bit on every disk; there would be no need for any other files or programs (but the configuration specification would be very large!).
- If every machine were to be identical and the configurations were never to change, the specification could be trivial: "Every machine consists of all the files in my software repository."

In practice, our machines are different, and they change; the configuration specification determines those aspects that we care about—usually things which may vary between machines or may change over time.[2] Many things are invariant, however, and there is no need to specify every detail of these; indeed, doing so will make the specifications confusing and unmanageable.

For example, the repository may contain several different versions of an application, and the configuration specification may simply define which version is installed on each machine. As a more complex example, it is probably not necessary (or even desirable) to use the configuration system to specify every parameter in the sendmail configuration file, although there may be a few parameters that must vary from machine to machine. In this case, a template file could be supplied from the repository, and the configuration specifications could be used to provide different values for substitution into the template for the different machines.

The decision about the level of detail contained in the specification will depend on the individual site and will vary over time; for example, it may be initially appropriate to install the same syslog.conf[3] file on every machine, but we may later decide that we need some variation, so we may switch to the use of a template. One day we might decide we need so much logging flexibility that we need to generate the whole file from data in the specification.

2. It might also be necessary to include information in the specification, rather than a fixed file, if the information needs to be deduced from other configuration information, particularly if this information comes from multiple sources ("aspects"). See section 1.3 for a discussion of this.

3. syslog.conf is the configuration file that specifies where to send the logging information from various different processes.

At the author's site, a typical machine includes about 400,000 system files (9GB), installed from about 1500 packages. The complete configuration specification for a machine consists of about 7000 parameters (1MB of XML).

Although copying files onto machines and customizing these files are important functions, they are relatively straightforward (see chapter 4). The real configuration problem is concerned with transforming the *high-level* requirements into simple *low-level* specifications. The high-level requirements specify the overall configuration of the system (e.g., a complete DNS service), while the low-level specifications determine which files to install on which machines, and how to modify them so that the machines work together to perform the overall function given by the specification.

Chapters 2 and 3 are concerned with the configuration issues, and chapter 4 covers the file distribution problem.

1.3 Complicating Factors

Although the basic statement of the system configuration problem is straightforward, there are a number of complicating factors which make practical solutions very difficult. Even developing sound manual procedures is not easy, and few automated tools address any of the following issues in a totally satisfactory way—note that sheer scale itself is not a significant problem; in general, it is complexity, in various forms, that causes the real difficulties. The following sections cover these issues in more detail.

1. *Managing relationships*—Configuring and maintaining the relationships between machines is harder than configuring individual machines in isolation.
2. *Managing change*—Both the configuration requirements and the physical systems are in a constant state of change.
3. *Managing diversity*—Managing many similar machines is comparatively simple; complete management of a site involves a diversity ranging from laptops to database servers, and supporting this diversity is much harder.
4. *Devolved management and "aspects"*—Many different people (and automatic systems) are involved in specifying the configuration requirements of a large installation. These requirements need to be consolidated and conflicts resolved without human intervention, wherever possible.
5. *Distributed systems*—A computing installation forms a distributed system, and the configuration problem involves all the difficulties associated with distributed programming, including communication, failure recovery, and latency.
6. *Usability*—People with a wide range of experience and ability will be expected to interact with a configuration tool, at several levels, and this poses special difficulties of usability.
7. *Autonomics*—Autonomic systems (which can perform automatic fault recovery) must be able to reconfigure without any human intervention. This places particularly strong requirements on the configuration system.

8. *Uncertainty*—In a large enough system, there will almost always be some hardware or software that has failed at any one particular time. Due to the latencies involved, it is not possible for a configuration tool always to have an accurate view of the entire system state. Configuration tools must be prepared to work in this uncertain environment.
9. *Security*—A good configuration system will be capable of reconfiguring every aspect of an entire site. This makes configuration tools an attractive target for malicious attacks, yet such tools are extremely difficult to secure.

1.3.1 Managing Relationships

System configuration, as defined above, is concerned with the overall functionality of a complete site; at a level above the configuration of individual machines, this involves understanding and managing the many complex relationships between individual configurable entities. For example, to provide a Web service, we need to configure more than just the Web server itself; we need corresponding entries in the DNS, we need appropriate configuration of the firewall, etc. (see section 1.4). Some relationships exist between different components on the same machine, but the relationships between components on different machines are the most challenging to manage, because of the increased complexity and the practical difficulties of working with a distributed system.

Most existing tools do not address this problem well, if at all. They are concerned with the configuration of individual objects; they may provide facilities for grouping and automatically configuring many such objects, but they do not "understand" the relationships between them. As an example, consider the configuration of an NFS server and its associated clients: a tool may allow us to configure the fstab[4] on multiple client machines with a single statement. It may also allow us to configure the exports file[5] on the server. However, it is important to ensure that the file systems the clients expect to import are actually the same ones as those being exported by the server. Most sites currently need to manage such higher-level relationships manually, and mismatches in these are a common source of configuration errors. Automatic configuration of these relationships is also necessary to perform any kind of autonomic fault recovery.

The client/server example described above is typical of the most common relationship type: one object has configuration parameters which must correspond in some way to configuration parameters of some other object. There are, however, other types of relationship which are also important. Static dependencies are a common but relatively straightforward issue—for example, the installation of one package may have prerequisite conditions on other packages. Dynamic dependencies are more difficult to manage. Reconfiguring a related set of services in the correct order to avoid inconsistent transient states, for example, is a difficult problem (see section 3.3).

For a tool to "understand" these relationships and provide support for their mainte-

4. The client configuration file specifying which remote file systems to mount.
5. The server configuration file specifying which file systems to export.

nance, some kind of higher-level model of the system is required; in the NFS example above, it is insufficient simply to manipulate the configuration files (fstab and exports) as opaque entities. It is even insufficient for the tool to manipulate individual lines in these files; it must understand the "meaning" of the configuration data they contain. Sanjai Narain et al.'s paper [59] gives a good example of how complex these relationships can become in practice, and section 3.1 looks at the different levels of modeling necessary to represent them effectively.

1.3.2 Managing Change

The rate at which configuration requirements change will clearly vary between sites—a research or academic environment is likely to have a model of change management very different from that of a production trading floor. However, there are many reasons why configuration changes may be necessary, and some are more urgent than others. For example (in descending order of urgency):

- Some critical hardware or software system has failed, and the system configuration needs changing to bring a replacement online.
- A critical security update needs applying.
- New application functionality (upgraded software) is required.
- The infrastructure is being refactored (e.g., changes to IP subnets or file servers).
- New or upgraded hardware is being introduced.

In conducting case studies, it is very difficult to determine the real change requirements of any particular site. Sites with configuration tools that cannot handle high rates of change often deny the need for such changes. However, sites that do have the capability usually exhibit comparatively frequent changes. Figure 1.2 shows the configuration change rate for a typical two-week period at the author's site. The short bars (and corresponding figures in parentheses) show the number of edits to systemwide configuration files (typically 10 to 100 files per day). The large bars show the number of resulting host reconfigurations (out of about 1200 hosts).[6]

Sites without the tools to manage such frequent configuration changes will often make architectural choices to minimize the need for change. For example, mounting applications from a remote fileserver avoids the need to update the applications on the client hosts. Other choices, such as the use of UPnP or DHCP to specify a NTP server, allow parts of the configuration problem to be delegated to other tools. Although this might appear to be a good strategy, it disperses the configuration information and makes it difficult to extract a coherent view of the entire system.

In addition to restricting architectural options, genuinely low change rates usually imply deferring and restricting the "lower priority" changes; this clearly has some impact on the service to the end user and on the efficiency of system management.

6. Note that these figures do not include minor software updates (probably on the order of 10 packages per day), which occur without explicit configuration changes.

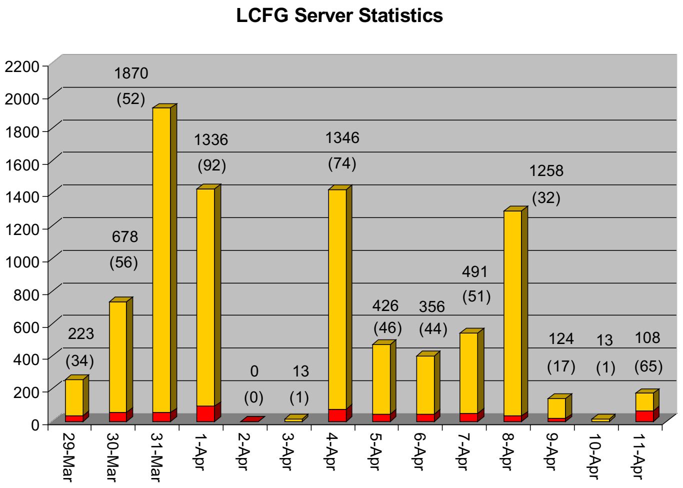


Figure 1.2: An example of configuration change rate

However, there is often a genuine requirement for different levels of stability for different applications: an internal system used for benchmarking and testing may require a very stable configuration, while an externally visible server will require rapid response to security updates. Providing this flexibility is difficult because it is necessary to ensure that the state of each machine remains consistent, even when it implements only a subset of the "latest" configuration changes.

1.3.3 Managing Diversity

Supporting very diverse configurations has many similarities with supporting a high rate of change. Sites often restrict the diversity deliberately to ease these problems. For example:

- It is common to see sites with automated systems for managing desktop workstations or cluster nodes (which are all very similar), but it is less common to see complete (*proscriptive*) management of servers or laptops.
- Sites deliberately restrict the diversity of the supported hardware, for example, or the degree to which workstations may be personally customized.

As with the change rate, sites with a capability for supporting diversity tend to exhibit a wide variety of configurations, while those without this capability are not always aware of the restrictions under which they are operating. For example, fully autonomic site management requires automated management of all types of node. Managing servers manually also exposes the most critical services to manual configuration errors and to potential difficulties with rebuilding in the event of hardware failure.

However, it is worth noting that supporting very diverse configurations makes testing difficult; some core configuration change may have a different effect on each of hundreds of different machines, if they have slightly different configurations. For this reason, diversity may be deliberately restricted in critical environments (such as on a trading floor).

1.3.4 Devolved Management and "Aspects"

Individual administrators have traditionally held responsibility for the configuration and management of individual servers. This requires a lot of general knowledge and the ability to coordinate (usually informally) with people managing related machines. As systems become larger and (particularly) more complex, this model becomes less tenable. Administrators need both to become more specialized and to deal with *aspects* of the whole site configuration. Responsibility for the different aspects of a machine configuration may *devolve* to a number of different specialists, often without very close contact between them. For example, a site may have a mail specialist who is responsible for the configuration of the mail "subsystem" on all the machines. A network/security specialist may be responsible for the site's firewalls, the iptables configuration on all of the hosts, and perhaps the routing.

Many of these aspects are now so complex that it is not possible for the administrators at a small site to understand and manage them all effectively. In some cases, some aspects are best handled by an external specialist who can be shared by several sites; for example, a university department may manage its own machines but allow the mail configuration to be defined by the central computing organization. Some applications, such as Grid computing, also require coordination of configuration specifications between widely disparate sites (see [25] for a discussion of this topic). In these cases, some aspects of the configuration (e.g., package versions) may need to be determined externally. It is not hard to imagine much more complicated instances of aspect devolution; for example, customers may be allowed control over the configuration of some aspects of a service provided by their ISP. Conversely, the ISP may be allowed control over some aspects of a customer site, such as external routing. Ultimately, the configuration of any one machine may depend on many different aspects defined by many different people from different organizations.

Even within an individual site, it may be necessary for end users to have control over some aspects of their desktop machines (e.g., versions of certain packages or the details of personal peripheral devices) without allowing them full control of the complete configuration. It is a common mistake for users to assume that the use of a centralized configuration management tool implies that they will have no control over their own machines; a good tool should actually provide them with a lot of flexibility and power in determining their own configuration. The degree of control users are permitted is, of course, a policy issue, and a good tool will give the administrator a lot of flexibility in setting the desired policy.

The overall configuration of an individual machine now needs to be computed by *composing* the aspects which may have been specified by many different people. It is important to note that these aspects do not usually correspond directly to any configuration files or subsystems on the target machine; one aspect may impact many different configuration files, and one configuration file may depend on many different aspects. For example, the inetd.conf[7] file would typically be affected by many different aspects,

7. inetd.conf is a configuration file which specifies services to be run in response to incoming requests.

not only the aspects defining the individual services but also those defining global policies on logging and access control; these overlapping aspects, determined by people with differing concerns, may need to be reconciled.

Clearly, it is possible for different administrators to specify aspects which imply conflicting configuration requirements on some individual machine. If the configuration is not proscriptive (i.e., some aspects of the machine have been manually configured), this presents another opportunity for conflict. In a highly devolved environment, it is not practical to negotiate every potential conflict manually, and it is an important function of a configuration tool to provide some support for conflict resolution in the composition process. The Arusha project [51] has the explicit goal of enabling this type of "federated" management. In most cases, however, conflicts are resolved by some override mechanism which gives some aspects priority over others. In practice, this is insufficient, and the resulting configurations may be unpredictable.

1.3.5 Distributed Systems

The mesh of services (DHCP, DNS, LDAP, etc.) which support the infrastructure of a typical modern site form a complex distributed application. Furthermore, these services must be extremely robust, since the failure of any one is likely to affect the entire infrastructure, often to the point where the configuration system cannot recover without manual intervention.

The configuration parameters for each individual machine at a particular site are computed (manually or automatically) from some overall specification of the site requirements. The configuration of each machine must then be *deployed* by transmitting it to the machine and modifying the actual machine configuration to match the specification. This deployment process entails all the problems of any distributed application; the configuration changes must be deployed on the individual machines reliably and in a suitable order, all in the face of possible failures and latency, both of the machines themselves and of the network. An ideal tool needs to provide some atomicity guarantees to ensure that the configuration is not left in an inconsistent state when failures or configuration errors occur. Section 3.3 discusses typical deployment techniques, and section 6.3 looks at some of the current research into novel approaches.

An important, but separate, question concerns the balance between centralization and decentralization of the configuration information itself. Clearly, there needs to be some single "central" specification of the overall purpose of a particular site at a sufficiently high level: "providing a student lab of 100 machines running Java," for example. However, as we have seen in the discussion of "aspects," the detailed components which make up this specification naturally come from many distributed sources. It is possible to collect this information together on a central server, which then computes the configurations of the individual machines and manages their deployment. Alternatively, the configuration information can be managed in a more peer-to-peer style, so that there is no single point which has complete "knowledge" of all the configuration details. Both of these approaches have advantages; robustness and scalability favor a decentralized solution, but this makes it very difficult to gather, for example, the distributed information necessary to manage relationships. LCFG (section 5.2) is a system which tends towards a centralized approach, and SmartFrog (section 5.4) tends towards a more distributed approach. Reference [23] describes an attempt to combine the advantages of both approaches.

1.3.6 Usability

Mark Burgess defines system administration as “the design, running, and maintenance of human–computer systems” [34]. This rightly emphasizes the role of the human administrator and his or her interaction with the system—in surveying configuration tool usage, “usability” frequently appears as the major barrier to the adoption of more powerful tools. Misunderstandings about tool usage are also a significant source of exactly those configuration errors which the use of the tool is intended to avoid. The configuration problem is inherently complex, and there are a number of good reasons why it is difficult to build tools that are clear and easy to use. However, usability should almost certainly be a higher priority objective of tool design.

As was noted in the Introduction, tools that manage the high-level configuration of a whole site require a very different approach from the traditional “bottom-up” process involving system administrators hand-editing configuration files. Administrators may have spent many years learning how to manipulate these files, and the formats are reasonably standard, so documentation and peer assistance are widely available. Many configuration tools tend to replace this familiar environment with completely different procedures, using languages and formats which are not widely known and which differ between tools. This presents a frustrating learning curve for most people and makes it difficult to share knowledge or move between sites. A further complication is that many different people, with differing skill levels and requirements, need to interact with the tool. For example:

- A technician replacing a faulty machine needs to specify the MAC address of the new box.
- A junior system administrator, managing a group of machines, needs to add a new machine with the same configuration as the existing ones.
- A system administrator managing a Web service wants to add some machines to act as accelerator caches. This involves changes to a number of related machines and services, similar to those described in the example in section 1.4.
- A senior system administrator wants to enforce a policy that every Ethernet subnet will include at least two DHCP servers.
- An administrator/developer is introducing a new application or service and needs to write code to interface the service to the configuration system.

The last example is particularly important; many benefits of a high-level configuration tool can be realized only if the tool is used in a proscriptive way—i.e., there are no parameters of the system that are configured manually (see section 2.5 for a discussion of this topic). This means that any new service or application which requires configuration management must be interfaced to the configuration tool. There will rarely be dedicated developers available, and the system administrators who are likely to perform this task are not usually in a position to learn large and complicated APIs in various languages; a very simple *scripting-based* interface is almost always preferable (and, normally, adequate).

Graphical user interfaces (GUIs) often provide a more usable interface to a system, particularly for naive users or casual use. However, it should be clear from the examples above that GUIs are not the whole answer in this case. Certainly a graphical interface is a big advantage for the lower-level tasks; in the first two examples listed above, the users would probably benefit from a GUI which would make these well-defined tasks easier to learn and less error-prone. However, as the specifications of intent become more complex, GUIs become less appropriate; it would probably be possible to specify the third example (Web caches) via a GUI interface, but the clarity of the intent would be completely lost in a myriad of dialog boxes and forms. Special-purpose languages for describing configuration specifications are the key to usability once those specifications become more complex; it is important that the high-level intent is clear and unambiguous (we discuss these important language issues more fully in section 3.2).

1.3.7 Autonomics

The term "autonomic" originates from physiology, where it is used to describe involuntary actions which occur without conscious control. The term has been used more widely in computing following IBM's "vision" set forth in their Autonomic Computing Manifesto [17]. A better explanation of this is given in [54], where Jeffrey Kephart says:

> Just like their biological namesakes, autonomic systems will maintain and adjust their operation in the face of changing workloads, demands and external conditions, and in the face of hardware or software failures of innocent or malicious origin.

This is, of course, not a new concept; Mark Burgess, for example, has advocated a similar principle in system configuration for some time [33], and cfengine (see section 5.1) incorporates the philosophy of a *self-healing* system. However, as we move towards managing configurations at a higher level, supporting autonomics introduces specific complications.

The aim is for an entire system to exhibit autonomic behavior, and this involves all possible levels; individual machines must independently attempt to adapt and recover from failures, but when they do not succeed, higher-level autonomic behavior needs to be invoked; for example, individual nodes may recover from some failures by restarting failed processes or "fixing" corrupt configuration files. However, if entire machines fail, replacements must be automatically reconfigured to take over their role. In an extreme case, an overloaded service might automatically negotiate with some other organization to temporarily make use of remote compute power (this is the realm of Grid computing).[8] Successful autonomics therefore depends on the existence of good solutions to most of the other issues described in this section (*no* manual intervention is possible). The problems of effectively decentralizing the configuration must also be solved to support effective and rapid configuration changes on an impaired infrastructure; section 6.3 describes some of the current work in this area.

Autonomics also has particular implications for the way in which configurations are specified (and hence for configuration languages). Specifically, the autonomic system must be able to change the configuration, so the administrator must not specify this in too much low-level detail. For example, we might declare:

- Servers X and Y export a particular filesystem (both providing read-only copies of the same data).
- Machine A imports the file system from server X.
- Machine B imports the file system from server Y.

(Perhaps this is intended to balance the load between the two servers.)

However, if server X fails, machine A will also fail, since its filesystem will disappear. An autonomic tool will be unable to reconfigure machine A to bind the other server, because the bindings have been specified explicitly and such a reconfiguration would violate the specification.

Existing fault-recovery systems would typically use event-condition-action (ECA) rules to specify recovery procedures in addition to the above specification, perhaps something like:

- If server X fails, change the specification so that machine A imports the filesystem from server Y.
- If server Y fails, change the specification so that machine B imports the filesystem from server X.

This is not a good solution; even in the trivial example above, the intent is not immediately clear, and we no longer have an explicit specification of the desired behavior (our original specification is going to be modified by the fault-tolerance rules). A much better specification[9] (from the autonomic point of view) would be:

- Servers X and Y export a particular file system (both providing read-only copies of the same data).

8. Of course, the configuration of the remote compute nodes must be compatible, and the site requesting service may want to pass on some configuration requirements; this is a good example of "federated" configuration, which has been explored further by the OGSAConfig project [3].

9. The preferred specification is *declarative* because it clearly states what we want to be true, without specifying how it is obtained. The previous specification is not declarative, since it specifies actions or procedures rather than just the required end result.

- Machine A imports the file system from server X *or* server Y.
- Machine B imports the file system from server X *or* server Y.

We are now leaving the choice of server up to the configuration tool. If one server fails, the tool will have the freedom to rebind all the clients to the working server, and we have the desired autonomic behavior. In practice, we would want such a tool to accept more high-level constraints as well, to ensure all the required properties. To reestablish (and maintain) the load-balancing behavior, we might add, "All working servers should have the same number of clients (plus or minus one)." At present, few languages support this type of loose, constraint-based specification. There is also a major acceptance problem, since most system administrators would require a good deal of trust in a tool before allowing it to automatically make such drastic reconfiguration decisions unsupervised.

1.3.8 Uncertainty

System configuration naturally takes place in an uncertain environment. In any sufficiently complex system, there will always be hosts or network connections that have either completely failed or are operating incorrectly. One important function of a configuration tool is to reconfigure the overall system to counter this type of failure. The configuration process itself must therefore be sufficiently robust to function in the face of uncertainty; no tool (or person) can *guarantee* to be able to deploy a configuration change onto a remote host. Furthermore, it is not always possible for the tool (or person) to know with certainty whether or not a particular configuration change has been successfully deployed (there may be failures in the monitoring process).

A fully autonomic configuration tool must be sufficiently robust to continue operating in the face of serious system failures, and it must be capable of reasoning about uncertainty in the deployed configurations.

1.3.9 Security

A powerful configuration system is the perfect vector for malicious software; a single change to some configuration specification can install software or modify the configuration on every machine at a particular site. In addition, several inherent characteristics of configuration tools (see below) make them notoriously difficult to secure. Most sufficiently powerful configuration tools are unable to address the security issue adequately, and the lack of standard software and procedures is probably the only reason why there have not been more serious exploits of this vulnerability.

The design of the UNIX operating system means that configuration of most subsystems requires root access. It is therefore difficult to partition the configuration tool so that vulnerabilities in one module do not affect the entire configuration. Developments such as SELinux are capable of providing the technology to solve this partitioning problem, but low-level system configuration operations have large and complex webs of dependencies which are difficult to manage; for example, the ability to reconfigure a primary DNS service or a DHCP service has wide-ranging security consequences, which are difficult to enumerate and contain.

In addition, it can be difficult to determine the origin of any particular configuration parameter. In a devolved management context, the final value of many configuration parameters is determined by composing input from several people, possibly in a distributed way, on several different machines. This implies some degree of host-based as well as user-based trust, which makes it impossible to simply use digital signatures to authenticate the configuration information.[10]

1.4 A Configuration Example

The following sections provide a concrete example of a fairly sophisticated configuration tool (LCFG)[11] being used to perform some typical configuration tasks—in this case, managing an externally visible Web server at the author's site. This involves:

1. Specifying the required configuration.
2. Physically installing the machine with that configuration.
3. Subsequently changing the configuration requirements.
4. Replacing failed hardware (possibly with hardware of a different type).
5. Decommissioning the machine and removing all traces from the network.

Note that many of these operations entail a chain of related configuration actions which affect not only the server itself, but other hosts on the network as well; the server must be loaded with the correct software and configured to run the appropriate services, but other hosts, such as the DNS servers, routers, and backup servers, must be reconfigured to support this service.

The following is intended to provide a complete example of a realistic configuration task and to show how it might be managed in a highly automated way. It illustrates a number of the general issues raised earlier in this chapter.

1.4.1 Specification

1. We create a DNS entry for the new machine. This site has chosen not to manage the DNS database with LCFG (although it could).

All the following steps are performed by editing a single configuration file on the LCFG server:

2. We specify that we want a "standard server" with a particular (supported) OS. Someone else will have defined what this "standard server" looks like, including the default software, default access rights, and all other configuration parameters; we simply specify the appropriate class.
3. We specify what hardware we are using. This determines any special drivers.

10. The configuration system probably has control over the distribution of the public key material as well.

11. For a more detailed discussion of LCFG, see section 5.2.

4. We specify which part of the network we want to connect the machine to. This determines how the routing works, which NTP servers get used, etc.
5. We specify that we want to run Apache (this does not happen by default at our site).
6. If the server is similar to an existing one, there may already be a suitable template for an httpd.conf file in the repository. If not, we create one and add it to the software repository as an RPM. There is no technical reason why the entire configuration file could not be specified directly in the LCFG specification file. However, we do not do this, for the reasons discussed in section 1.2.
7. We might specify parameters to plug into the template.
8. We specify the MAC address for the machine. This information will be *aggregated* into the configuration information for the DHCP server, so that we automatically get DHCP service; we do not have to maintain the DHCP server configuration independently.
9. We specify that we want http(s) holes in the firewall. This information will be aggregated into the configuration for the host that manages the firewall router.
10. We specify that we want an SSL certificate. An LCFG component will automatically generate and sign a certificate at install time.
11. We specify any host-specific parameters—perhaps we want to specify a non-default disk layout, for example.
12. We might specify that we want our data mirrored. This information would be aggregated into the configuration for the mirror server.

It is worth observing how some of the important principles from the previous section are reflected in this example:

- Several important inter-machine relationships are established automatically; for example, the MAC address is passed to the DHCP server configuration, and the firewall hole requirements are passed to the firewall—we do not need to maintain these relationships by manually reconfiguring the servers.
- Any degree of diversity is easily supported by combining the desired "aspects" and customizing them with any necessary machine-specific parameters.
- There is a high degree of devolved management; aspects such as the configuration of the base operating system, local site policies, hardware specifics, and network configuration are all delegated to others—we do not need to be concerned with these at all. In many cases, these aspects will affect common sets of parameters, and the tool is responsible for resolving any apparent conflicts.

```
#include <lcfg/os/redhat9.h>
#include <lcfg/opts/server.h>

#include <lcfg/hwbase/dell_optiplex_gx240.h>

#include <inf/sitedefs.h>
#include <inf/wire_c.h>

!boot.services mADD(lcfg_apache)

!profile.packages mEXTRA(my-apache-config-*-*)

dhclient.mac 00:06:5b:bf:88:7e

#include <inf/ipfilter.h>
ipfilter.export https

!x509.keys mADD(myweb)
x509.service_myweb myweb.inf.ed.ac.uk

!fstab.size_hda1 mSET(10000)
!fstab.partitions_hda mADD(hda3)
!fstab.mpt_hda3 mSET(/webdata)
!fstab.size_hda3 mSET(free)
!fstab.type_hda3 mSET(ext3)
```

Figure 1.3: An LCFG configuration specification

Figure 1.3 shows what this configuration *profile* (specification file) would look like in practice; unfortunately, LCFG does not score very highly on usability, and the syntax is notoriously obscure. However, the details are not particularly relevant here, and very often a specification such as this would be created by copying and editing the profile for some similar machine.

1.4.2 Installation

Server installation is a technician-level task. After physically connecting the machine, it is booted using PXE (or CD), and installation is fully automatic; the corresponding configuration specification is determined (ultimately) by relating the MAC address of the machine to that specified in the configuration profile.

In our case, Kerberos host keys and SSL certificates will be generated automatically at install time, and this operation will require an authorized administrator to provide Kerberos credentials at some point during the installation. Otherwise, the install is unattended.

1.4.3 Changing the Configuration

LCFG can support a high rate of configuration change; configuration changes only require an edit of the profile. If this changes, then the LCFG server will propagate the new configuration to the machine, and all the affected components will reconfigure automatically. Similarly, changes to any of the included aspects will trigger reconfiguration of all affected machines; this can be clearly seen in figure 1.2 where some file edits led to very large numbers of host reconfigurations.

1.4.4 Replacing Failed Hardware

It is trivial to replace a failed machine with similar hardware; the profile is edited to specify the new MAC address, and the new machine will install as an exact replica of the original one when it is booted. If the replacement hardware is different, it is only necessary to change the hardware type in the profile and perhaps some details such as the disk partitioning. This ability to substitute machines with slightly different specifications is an important advantage over simpler recovery techniques such as system backups or filesystem cloning.

The reinstallation does not restore the user data (Web content, in this case). This must be reinstated manually from the backups or the mirror server.

1.4.5 Decommissioning

When the server is decommissioned, the profile is simply deleted, and the related services will automatically reconfigure; the holes will be removed from the firewall and the MAC address from the DHCP server. In our case, we also need to delete the DNS entry separately.

1.5 Some Key Points

- Configuration specifications are fundamentally *declarative*—they specify the configuration requirements, not the steps necessary to transform a system into compliance with those requirements.
- The job of a configuration tool is to make a physical system conform to the declarative specification, and to maintain this conformance as the environment and the specification change.
- The "physical system" refers to an entire installation, and managing the relationships between nodes usually presents more difficulty than managing the individual nodes.
- Diversity and change are ubiquitous; they need to be embraced rather than avoided.
- Many different people will be involved in various aspects of a system configuration. Tools need to take into account the different perspectives (and levels of experience) of these users and handle potential conflicts automatically whenever possible.
- Configuration systems must operate in an unreliable environment. It is not always possible to be certain whether a desired configuration has actually been deployed.
- Support for autonomics requires the ability to perform major reconfigurations without manual intervention.
- Configuration is not simply "scripting." New paradigms, languages, and working practices are necessary to fully address the above issues.

2. Approaches to Configuration Management

There are currently no tools available that address the full scope of the configuration problem, as described in the previous chapter; indeed, a full solution requires answers to problems which are still in the realm of research (some of these are discussed more fully in chapter 6). In practice, different sites have different priorities, and they adopt different compromises; the case studies described in [22] and [47] show some typical situations. However, it is possible to identify a broad hierarchy of approaches, starting with a low-cost, largely manual approach, and moving towards highly automated solutions which involve a considerable investment of effort.

Historically, the development of system configuration has tended to mirror the evolution of a typical site; starting with an entirely manual process, and maturing towards more control over the configuration, more automation, and more concern with higher-level properties. The individual steps in this process are not entirely well defined; they are not a strict progression, and most sites will use a mixture of techniques. However, there are several distinct stages each of which presents additional challenges and yields additional benefits:

1. *Manual configuration*—Individual machines are manually installed and configured.
2. *Cloning*—Multiple similar machines are installed by *cloning* the image of one hand-crafted machine. The images usually need customizing in some way, either by hand or by additional scripting, to differentiate them.
3. *Procedural scripting*—A program or script is used to create or modify the configuration (possibly following a cloning operation). This provides automatic support for more diversity or a greater rate of change.
4. *Declarative configuration*—A declarative language is used to specify some of the configuration requirements, and a special-purpose tool automatically computes and applies the necessary changes to bring the machine into compliance with those requirements.
5. *Proscriptive configuration*—The entire configuration of the machine is under the control of the configuration system. Machines can be recreated or duplicated, and all relevant aspects of their configuration can be changed via the configuration system.
6. *Modeling relationships*—The configuration tool manages relationships between machines automatically, preventing errors due to mismatches between servers and their clients.

7. *Deducing configurations*—Configurations are specified as looser “constraints,” rather than absolute values, allowing the system to reconfigure automatically in the face of failures and allowing multiple people to collaborate on the configuration without conflict.
8. *Autonomics and feedback*—The configuration system makes use of live feedback information to adapt to failures and maintain service levels in response to changing demands.

The majority of large sites are probably using some form of scripted configuration management. Manual configuration is usually suitable for one-off or very small situations, and cloning can be appropriate for certain applications such as clusters (see chapter 4). Fully proscriptive configuration is entirely practical, but many sites will be unprepared for the necessary commitment and discipline. The higher levels (see 7 and 8, above) are becoming increasingly important, since they are a prerequisite for fully autonomic services, but at present they tend to exist only in limited production applications or in research systems.

"Copy this disk image onto these machines"

⇓

"Put these files on these machines"

⇓

"Put this line in sendmail.cf on this machine"

⇓

"Configure machine X as a mail server"

⇓

"Configure machine X as a mail server for this cluster"
(and the clients will automatically be configured to match)

⇓

"Configure any suitable machine as a mail server for this cluster"
(and the clients will automatically be configured to match)

⇓

Configure enough mail servers to guarantee
an SMTP response time of X seconds

Figure 2.1: Levels of configuration specification

In some sense, this range of approaches moves from dealing with *low-level* concerns, such as disk images and files, toward *high-level* ones, such as services and service levels (see figure 2.1). This is analogous to the development of programming languages, where the emphasis has moved away from bits and registers, through variables and subroutines, towards the manipulation of higher-level entities and relationships. In both cases, the higher-level approaches require a firm low-level base, with well-understood theoretical properties, to provide a reliable foundation. Of course, the ultimate requirements of every site are expressed in very high-level terms (e.g., "I need a reliable multi-tiered Web service"), and this must always be translated into some low-level physical requirements; the issue is how much of this process can be performed automatically, and how much must be performed manually.

Alva Couch [39, 38] has postulated that sites tend to move up this hierarchy of control as they increase in size and complexity, often facing a difficult transition between phases. The case study from Argonne National Laboratories [45] is a good illustration of the social issues typically involved in adopting a new level of configuration technology.

2.1 Manual Configuration

Manual configuration usually involves an administrator hand-editing configuration files (or perhaps using a GUI interface) on each individual machine. In most cases where a professional system administrator is involved, the administration problem will be sufficiently complex and/or critical that this is impractical, and some degree of configuration automation will be more appropriate. However, in a few cases (e.g., non-critical one-off systems) completely manual configuration may be a reasonable solution. It is also a pragmatic approach when the cost of implementing and maintaining an automated solution is disproportionately high: for example, in very small or very diverse environments. It is common to see sites with large numbers of clients under automatic management and a smaller number of manually configured servers. The motivation for this is clear; the servers are more complex, and manual configuration is a viable option when their number is small.

It is useful to summarize some of the major problems with this manual approach, both to assist in evaluating this option and to act as a baseline against which to evaluate more automated approaches:

- It is difficult to ensure that configurations on supposedly identical machines are in fact identical (and that these configurations actually represent the intended configuration).
- Configuration changes are difficult to support because the effort is approximately linear in the number of machines. This has implications—for example, for security (security patches need timely applications).
- It is unlikely that the configuration can be restored easily if the hardware is upgraded or requires replacement; the configuration information is not separated from the rest of the operating system and cannot be reapplied quickly and reliably to a newly installed machine.[1]
- None of the higher-level configuration automation facilities (e.g., validation of relationships between machines, autonomics) are available.

1. If the machine type has changed, for example, then simply installing an image backup is probably not sufficient.

The combined effect of these factors generally leads to sites which are less reliable, more difficult to change, and less secure. Of course, the ongoing maintenance of the site also requires more manual effort.

2.2 Cloning

One of the earliest approaches to managing large numbers of machines was the concept of cloning. This usually involves hand-configuring some *golden-copy* machine and then replicating the entire file system onto a number of other machines, by either a file-level or a disk-level copying process. This does address the difficulty of creating a number of machines with a guaranteed identical configuration, and it does help to some extent in dealing with change; an entire cluster can be reconfigured by making a change to the golden copy and re-cloning every machine. (Of course, this is a very disruptive process, which might be acceptable as a scheduled activity in a compute cluster but would probably not be acceptable for desktop machines.)

Managing diversity of configurations is the most obvious drawback of the cloning approach.[2] When there are differences in hardware or differences in the required functionality of machines, pure cloning is not sufficient. Pure cloning nearly always evolves into a hybrid approach in which the cloning operation is followed by some scripting process to differentiate ("customize") the cloned machines (see, e.g., [53, 64]), and cloning alone does not address any of the higher-level issues.

Despite these limitations, there are a number of situations where cloning can still be useful; for example, it is an efficient way of installing the bulk of the system files onto a new machine, before the configuration system takes control. Cloning tools can also be pressed into service as a substitute for true configuration tools if the requirements are very simple; this is discussed further in chapter 4.

2.3 Procedural Scripting

The term *scripting* can be used to describe any situation where the configuration of a machine is modified by some program or script. The use of scripting usually arises out of a need to address the following problems:

- The configuration of the (possibly cloned) machines needs to vary, perhaps because of differences in the machines themselves (hardware), or because of differences in the required functionality.
- The configuration of existing machines needs to be modified to track changing requirements or to correct discrepancies that have appeared in the actual configuration.

In the first case, the scripting is being used to address the problem of diversity. Typically, a machine is created by cloning some generic template machine, and a script is then applied to modify the configuration depending on the specific characteristics and requirements of the individual machine. Alternatively, the script might operate on

2.Interestingly, this is one area that is not normally a problem when managing configurations manually.

a bare machine to build the entire system by selecting and loading specific software packages and generating configuration files.

In the second case, the scripting is being used to solve the problem of change (without the need to rebuild the entire machine). Note that there are two possible reasons for a mismatch between the *actual* and the *designated* configuration—either what we want the configuration to be has changed, or the actual configuration has changed from what we wanted it to be (presumably due to some human or system error).

Ad hoc scripting is a very attractive and popular approach to configuration management; additional scripts can be created incrementally to address pressing configuration problems, and it is easy to migrate slowly towards a higher degree of automation. Existing systems do not need to be rebuilt, and there is no jump in technology which might involve significant staff retraining and familiarization. However, as with any type of programming, an undisciplined approach to scripting can quickly lead to serious problems, and the following pitfalls are common:

- Incremental development often leads to poor structuring, and mature sites frequently have very large quantities of scripts, all dealing with essentially low-level concepts. The overall purpose of the configuration can become obscured, and it is difficult to have any confidence that the scripts reliably implement the intended high-level configuration requirements (or indeed, any idea of what the intended configuration actually is!).
- Maintenance is likely to be difficult and error-prone.
- Scripts that have the power to perform arbitrary actions on multiple machines, usually as root, can cause spectacular system failures (either through small errors or malicious intent).
- Collaborative development of configurations (devolved management) is difficult because of the potential conflicts between scripts created by different people.
- Arbitrary scripts in a rich language are a poor basis on which to layer high-level tools. For example, it would be difficult for a higher-level tool to take a collection of arbitrary scripts and determine which machines were intended to be mail servers.

Scripts are usually written in some familiar language such as shell or Perl; crucially, these tend to be procedural rather than declarative languages, and research work in configuration theory has highlighted some common practical problems that are a consequence of this approach. It is possible to write scripts in most languages that avoid these problems, but this requires considerable care and understanding. Tools designed specifically for configuration tend to use a declarative approach (see section 2.4) and should avoid these issues by design.

2.3.1 Idempotence

Actions are called *idempotent* if executing them multiple times will have the same effect as executing them once. If scripts do not satisfy this property, then it is necessary to keep very careful track of which scripts have been applied to which machines; in practice, this is extremely difficult.[3]

Add the following line to /etc/services: `amidxtape 10083/tcp`
Add the following line to /etc/services if it does not already exist: `amidxtape 10083/tcp`

Figure 2.2: Non-idempotent vs. idempotent actions

Notice that it requires great care to code such operations correctly using a general-purpose language; in this example (figure 2.2), even the (naive) idempotent version fails if we attempt to change the port number for the service (the file will end up with two lines for the same service).

2.3.2 Closures

If two scripts affect the same object, it is likely that different results will be obtained by running the scripts in different orders. This almost certainly represents a configuration error, but it is likely to go unnoticed in practice, except for the apparently random effect that it is likely to have on the configuration of the target machine. To avoid this, tools should aggregate the requirements from the different aspects of the specification into disjoint *closures*, resolving (or reporting) conflicts before generating a single action to modify the target object.

Add the following line to /etc/services if it does not already exist: `amidxtape 1234/tcp`
Add the following line to /etc/services if it does not already exist: `amidxtape 4567/tcp`

Figure 2.3: Conflicting actions

A good configuration tool would resolve the conflict implied by the example in figure 2.3 before attempting to implement the configuration (note the different values for the port numbers). Ideally, priority would be given to one of the specifications, based

3. It is also impossible simply to run the script again to fix a machine whose state is unknown or somehow corrupt.

on some logical criteria (not just the order of appearance of the specifications in some file). If this is not possible, the conflict should be reported for human resolution (perhaps the two different requirements were specified by different people).

2.4 Declarative Configuration

Declarative specifications describe the desired end result of a configuration rather than the process required to achieve that result (which should be computed by the tool).

The file /etc/services must contain one copy (only) of the following line: amidxtape 1234/tcp
Add the following line to /etc/services if it does not already exist: amidxtape 1234/tcp

Figure 2.4: Non-declarative vs. declarative specifications

The distinction between the two examples in figure 2.4 is subtle but important. The non-declarative version describes an action. The declarative version specifies the desired final state, and the tool is free to choose the appropriate action (which may mean doing nothing). Special-purpose configuration tools tend to use declarative specifications, and this has the following advantages:

- Because they do not suffer from the idempotence problem discussed above, tools can run continuously to monitor the actual configuration and *converge* it with the designated configuration whenever it deviates. This is very different from the application of a one-off configuration change action, which instantiates a configuration at one time, but does not guarantee that it will not diverge at some time in the future.
- Tools can reason about the configuration statements and detect conflicts before attempting to apply the configuration.
- The declarative statements are more amenable to automated reasoning at a higher level: for example, matching configurations of servers to the configurations of their clients, or reassigning whole services when servers fail.

Although declarative specifications should be much easier to formulate than their procedural implementations, system administrators are used to dealing with procedures, and sometimes have difficulty in expressing the desired end result without considering the method by which it is achieved. This difficulty is compounded by current declarative configuration languages, which are not so clear and well-developed as the familiar procedural programming languages.

2.5 Proscriptive Configuration

If a configuration system controls the entire configuration for a set of machines, with no manual intervention, then it is known as *proscriptive*. The pure cloning process, for example, is proscriptive (but, of course, is unsuitable for handling the diversity of a realistic site). Considerable effort is required to implement a fully proscriptive configuration for a diverse installation, and many sites do not have the necessary tools, or do not consider it worthwhile; in practice, most sites probably have proscriptive configuration for their clients, but not for their servers.

Non-proscriptive configuration processes are subject to configuration divergence if various aspects are moved in and out of the scope of the configuration tool (see [42, 40]); over time, the configuration of machines that are intended to be identical will tend to diverge in unpredictable ways. Of course, those parts of the configuration that are outside the control of the tool are also subject to all the problems of manual configuration noted above.

Proscriptive configuration tools are a prerequisite for machines to be completely reconfigured autonomically, perhaps in response to failures or changing external load.

2.6 Higher-Level Configuration

Once a site is able to perform declarative, proscriptive configuration automatically, a different set of problems appear; at this level, the physical *fabric* of the site can be completely instantiated and maintained from a "soft" configuration description. The focus of the problem then moves towards creating and manipulating that description. This is analogous to having created the hardware for a new processor (or perhaps the assembler) and starting to concentrate on programming the device to solve useful problems.

For example, the correct functioning of a site and the capability of performing true autonomics depend heavily on the ability to manage the relationships between machines; if a replacement server is automatically substituted for a failed one, then all the clients must be reconfigured to use the new server. Devolved management and autonomics also require a configuration system that has enough flexibility to compute an explicit instance of a configuration from a changing set of available resources and a changing set of high-level requirements, as well as taking into account external factors such as load. This requires configurations to be specified in a less explicit way; specifying all of the low-level details is no longer appropriate, since the system must have the freedom to choose these so as to satisfy the high-level requirements given the available resources. Constraint-based specifications allow the system to choose from *all* of the acceptable solutions.

Ultimately, it is the behavior of a system that is important, rather than the details of its implementation. Specifying this required behavior and having suitable implementation details computed automatically requires feedback on the actual performance and the ability to incorporate this into an overall reasoning system.

All of these higher-level functions involve complex manipulations of the configuration description, and they rely on solid lower-level tools with suitable interfaces to ensure the effective deployment and monitoring of the actual configuration. Production systems cannot yet support these higher-level functions in any general way, although chapter 6 describes some of the current research in this area.

2.7 Some Key Points

- The full configuration problem involves translating high-level requirement specifications into low-level configuration parameters (as well as actually deploying the resulting low-level descriptions).
- There are a range of approaches that automate increasingly higher-level stages of this problem; individual sites need to choose an approach suitable to their requirements and available resources.
- Necessity and economy of scale usually mean that large sites employ higher levels of configuration automation than small sites. This leads to corresponding benefits in terms of reliability, consistency, and security.
- Moving from one approach to a higher-level approach usually entails a significant change in mindset, and can often be a difficult transition.
- It is possible to instantiate and maintain physical configurations from declarative descriptions. Beyond this level, the configuration problem becomes one of manipulating these high-level descriptions to meet the necessary requirements.

3. Configuration Tools

Existing configuration tools approach the problem from many different angles, showing their different ancestry and emphasizing different aspects. However, it is possible to compare the various approaches in a number of important dimensions:

- The model which a configuration tool uses to represent the system.
- The language provided for users to describe this model.
- The way in which the specified configuration is physically deployed.
- The extent to which monitoring and feedback are used to ensure that the physical system matches the required specification.

Unfortunately, a lack of standards means that few tools are capable of useful interoperation, and it is usually impractical to mix tools in a way that takes advantage of their different strengths. Some kind of standard interface to the low-level functions would be particularly useful in allowing development of high-level tools which could build on existing implementations of the low-level operations (see [27] for a discussion of this).

3.1 Configuration Models

The *model* supported by a configuration tool determines the types of objects that can be described and the relationships and operations that can be represented. There is an important distinction between this and the language used to describe the model (e.g., multiple languages may be used to describe the same model).

Very simple approaches to the configuration problem (e.g., basic scripting) involve no special model at all beyond that of the underlying operating system; the tool deals with fundamental entities, familiar objects such as files and processes. The advantages and disadvantages are similar to those of programming at the assembly-code level, where the model is the hardware architecture of the physical machine. For example:

- In theory, it is sufficiently flexible to write code which meets any requirements.
- There are no new languages to learn or compilers to buy.
- The code is expensive to produce and maintain.
- Correctness and clarity are harder to achieve.
- Interoperability and code-sharing are very hard, because they rely on agreements about interfaces and standards.

At the opposite end of the spectrum, the *DMTF* (Distributed Management Task

Force [2]) defines a *Common Information Model* (CIM) and communication/control protocols (WEBM) [15]. This is intended to be a standard model that will allow configuration management tools to interoperate at a high level:

> It is not sufficient to manage personal computers, subnets, the network core and individual systems in isolation. These components all interoperate to provide connectivity and services. Information passes between these boundaries. Management must pass across these boundaries as well. [46]

Models such as this, which provide comprehensive, detailed standards, are probably necessary to address the highest-level goals of interoperability between diverse systems in an autonomic way. Although CIM is being used as a practical system management tool (see, e.g., [61]), development of CIM schema and the associated code is not within the scope of the average system administrator; they require considerable background information and understanding, and are largely intended for developers. Very few sites are likely to have CIM schema and interfaces for all the subsystems they need to manage, and this probably makes CIM-based systems unsuitable for most current sites, unless they have considerable development resources.

LCFG (see section 5.2) and SmartFrog (see section 5.4) are examples of approaches with an intermediate model; both define a simple framework of "components" with a mechanism for composing declarative descriptions of the configuration and passing them to the components for instantiation. Standard components (which sites may choose to use, or not) are available, but custom components can be created comparatively easily using shell, Perl, or Java. The frameworks support important generic features such as aggregation, which are difficult to implement in a more ad-hoc way, and the configuration information is managed as simple declarative structures suitable for high-level processing. This level of modeling appears to be a good practical compromise.

3.2 Configuration Languages

To some extent, a language for specifying configurations depends on the underlying model; for example, declaring file permissions appears to be quite different from declaring a constraint about the relationship between two services. However, ignoring the details of syntax, it is possible to make some useful generalizations.

As noted above, most special-purpose configuration languages have independently converged on a declarative approach—that is, the language makes statements about the designated configuration and the tool computes the necessary actions to achieve (and maintain) that configuration. General-purpose, declarative programming languages do exist, and these have been studied in a configuration context (e.g., Prolog in [41]). However, configuration languages tend to have other requirements (see below) which are not well met by these languages, so most configuration tools define their own language.

Prototyping appears to be ubiquitous as a method for structuring configurations and addressing the problems of collaboration and devolved management. Support for this is almost universal in all configuration languages; for example, one person will create a generic template for some particular aspect, and this will be *inherited*[1] and *specialized* (perhaps several times) for some specific purpose, by overriding some of the default values. Figure 3.1 shows an example in the SmartFrog configuration language (see section 5.4).

```
Disk = {
filesystem = "NTFS";
size = 20;
}

// A bigger disk is a disk augmented with a size 40
BiggerDisk = Disk {
size = 40;
}

// An HP is a bigger disk is augmented
// with an attribute "make"
HPDisk = BiggerDisk {
make = "HP"; }
```

Figure 3.1: Inheritance in SmartFrog

Current research is looking at more general ways of composing values from different aspects (see section 6.2) to try to address some of the deficiencies of this rather naive process.

As with programming languages, support for structuring the configuration into meaningful subunits which can be inherited and reused is also important. Unfortunately, this is not a well-developed aspect of many existing configuration languages, which tend to have evolved in a bottom-up manner; however, SmartFrog, for example, provides a hierarchical component model which supports good reuse at multiple levels, while LCFG provides structuring at a single-component level.

Addressing the higher-level aspects of the configuration problem places extra demands on a configuration language: for example, the ability to represent relationships and constraints in a meaningful way. Such facilities are not yet a major part of any production tool.

Finally, as was noted in chapter 1, usability is extremely important; configurations are determined by many people with different skill levels, and configuration specification errors are a common source of serious failures. Most current tools perform very badly in this respect, and finding good ways of clearly specifying relationships and

1. Note that this is technically "instance-inheritance," as opposed to the "type-inheritance" familiar from object-oriented programming languages.

requirements is an active research area. It is worth noting that GUI interfaces are almost certainly not appropriate for anything beyond very simple specifications;[2] the Active Directory interface (see figure 5.3), for example, almost certainly obscures the relationships and intent behind related settings, and does not attempt to address higher-level issues such as constraints.

3.3 Deployment Issues

After a model for the desired configuration has been developed and has been expressed in an appropriate language, a configuration tool is needed to deploy the specification onto a physical system.

Most operating systems are completely ill-suited to automated configuration management; changes to the configuration on an individual machine may involve changes to several different files (in as many different formats) and often the restarting or notification of assorted processes as well. The details of this process are largely uninteresting, despite the fact that it represents the bulk of the code in most configuration tools; when the configuration specification changes, some code is required to translate the new specification into (possibly) new configuration files and perform any additional actions, such as restarting daemons. The real difficulty in deployment comes in managing the relationships between machines and in maintaining those relationships in the face of changing requirements and uncertain hardware.

3.3.1 Centralized vs. Distributed Models

The most straightforward approach to configuration deployment is for a central server to have full knowledge of the requirements and the available resources (this is the model used by LCFG). The server uses these to compute the low-level configurations of each of the machines in its domain, and it deploys the resulting configurations directly.[3] If the specification changes (or a hardware failure occurs), the server has all the necessary knowledge to recompute and deploy a new configuration. This is a very attractive approach, since the server (theoretically) has sufficient knowledge to validate configuration relationships in advance and to control the overall configuration of the site. However, as with a central compute server, it presents a number of problems:

- The computation load on the server means that recomputing configurations for large numbers of machines can take some time, and this is a common occurrence for systems that support good modeling of relationships. For example, when an NFS server is reconfigured, it may be necessary to reconfigure all of its clients. The graph in figure 1.2 shows the result of this effect on the number of recompilations performed by the configuration server. The resulting reduction in response time may make the system unsuitable for autonomic reconfigurations.

2. Of course, GUIs are very useful for presenting less-skilled staff with limited choices.

3. This implies that the server controls the configuration; the actual transport may be either a "push" or a "pull" operation.

- The source of the configuration information is not always centralized, and collating the information into a central location may not always be appropriate. The extreme example is the case of a laptop, which may be disconnected when a configuration change is required. In a federated management situation, it is also likely that different aspects of the configuration would be “owned” by different organizations.
- A centralized system can be replicated and hardened against failure, but there is still a significant risk of machines being unable to reconfigure due to network failure.

Section 6.3 looks at some of the research into alternatives approaches; peer-to-peer techniques are an obvious possibility, and these work well in specific applications, but providing a more general framework appears to be more difficult.

3.3.2 Change Sequencing

The declarative approach to configuration specification is very attractive; the specification defines the desired state, and the tool has the job of deploying that configuration. However, in practice, the process of deployment may take some time, and the configuration of the entire system will move through many intermediate states during the transition. Ideally, we would like all of the possible intermediate states to be “valid” in some sense; we certainly would not want the system to be left indefinitely in an invalid state if some operation failed, and normally we would not want transient invalid states to last “too long.”

As a simple example, consider the updating of a package on an individual machine —at some time during the process, there will be a period when the package is inconsistent because some of the installed files belong to the old version of the package and some belong to the new version. However, a good package manager will minimize this time period and ensure a truly transactional installation if at all possible (i.e., either the new package is installed correctly and completely or the old package is left unmodified). A more complicated example occurs when we wish to change a file server:

1. Deploy the new file server.
2. Change all the clients to point at the new server.
3. Withdraw the old server.

At each stage, we must wait for positive acknowledgment that the previous stage has completed.

This is clearly a procedural process, and if we simply change a declarative specification, there is no guarantee that a configuration tool will perform the transition in this orderly fashion. The ideal solution is to augment the declarative description with conditions that describe the valid and invalid states. In theory, a configuration tool should then be able to plan a sequence of changes between two configurations that moves only through the valid states (if possible). This again is an interesting area of research (see section 6.3).

A related problem occurs when some feature of a configuration change must be deferred for some reason; for example, LCFG will not change the configuration of the display manager while there is a user logged in to the console (because this would involve restarting the display server, which would be rather disruptive!). Similarly, a compute cluster node would probably want to defer a change to the maths library during a very long-running compute job. Allowing certain parts of an overall configuration change to occur while deferring others—e.g., installing a new version of some software with new configuration files without restarting the running daemon—has the potential for leaving the machine in an unexpected state which has not been validated and may cause problems. Alternatively, deferring all configuration changes when any one change is blocked has the potential for blocking all configuration changes for an unacceptable length of time.

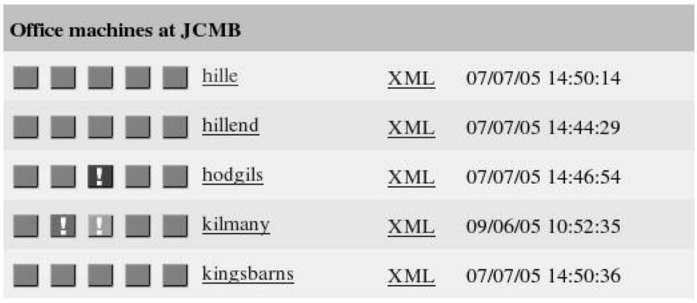
Deployment order of related services is an important issue for distributed applications such as Grid services. SmartFrog provides primitive components that allow other components to be deployed in sequence or in parallel and to construct complex deployment strategies, with reliable rollback in the event of failure.

3.4 Monitoring and Feedback

All of the previous discussions have been concerned with computing and deploying a designated configuration, which represents one of the many possible ways of configuring the overall system to meet the ultimate requirements. At any one time, there will be a discrepancy between this and the actual configuration of the system. In most cases, this discrepancy will be a normal consequence of the latency in the deployment process; in addition to the natural latency of the tool, changes may be explicitly deferred for the reasons described in the previous section, or machines may simply be unavailable at the time when a change is initially deployed. In a sufficiently large system, the actual configuration will never correspond exactly to the designated configuration, since the requirements are likely to change again before the entire system has implemented the previous change—the very appropriate term *asymptotic configuration* is used to describe this situation. Any monitoring system is also subject to latency and system failures, so it is theoretically impossible to be absolutely certain of the actual state of the system at any one time.

A simple monitoring tool will provide feedback to the system administrator on the apparent state of the configuration deployment; for example, figure 3.2 shows part of an LCFG status display indicating which machines have implemented their current designated configuration.[4] In a simple situation, this discrepancy can be monitored manually, and the system administrator will make a decision about which discrepancies are due to acceptable latency and which represent a true failure. Of course, in complex situations, some automated help is necessary to determine the root cause of the prob-

4. The machine Kilmany has apparently not acknowledged configuration changes for several weeks (it is probably turned off).



Office machines at JCMB			
■ ■ ■ ■ ■	hille	XML	07/07/05 14:50:14
■ ■ ■ ■ ■	hillend	XML	07/07/05 14:44:29
■ ■ ! ■ ■	hodgils	XML	07/07/05 14:46:54
■ ! ! ■ ■	kilmany	XML	09/06/05 10:52:35
■ ■ ■ ■ ■	kingsbarns	XML	07/07/05 14:50:36

Figure 3.2: The LCFG status display. (N.B. Dates are UK format.)

lem; for example, the failure of a particular router may generate apparent configuration failures in a large number of machines. This situation is not specific to configuration tools, and the problems of monitoring and "root cause analysis" are normally outside the scope of system configuration.

However, an autonomic system must be capable of automatically reconfiguring a whole group of machines to compensate for failed nodes and even partial network or service failures. This implies some automatic analysis of the monitoring information to determine the cause of failures, followed by a logical deduction and deployment of some alternative solution. Although many subsystems exhibit this type of behavior in a limited domain, the general problem is very hard, and there are no comprehensive solutions. Reference [23] describes an experimental approach to a more general solution.

3.5 Some Key Points

- Standards defining sophisticated models are necessary for interoperability between sites and tools, but these are complex to work with and not yet widely implemented.
- Tools that impose no model are inadequate for higher-level configuration and make it difficult to share and collaborate on configuration management.
- There is great variety in configuration languages, but declarative languages with support for prototyping appear to be particularly appropriate.
- Peer-to-peer technologies for configuration management have the potential for offering some important benefits, but they also present (as yet) unsolved difficulties, and a centralized approach to configuration management is probably more practical at present.
- Sequencing of related changes between components is not well handled by most configuration tools, although work on this is being driven by the requirements of distributed Grid applications.
- Monitoring of configuration states in a form suitable for human consumption is common. Processing this information to provide feedback for autonomic reconfiguration is much harder.

4. File Distribution and Package Management

File distribution and installation are only a small part of the full configuration problem; computing which files to install on each node and managing the semantics of the small proportion of "configuration files" (perhaps 0.005%) are the real difficulties. For true configuration tools, package management and distribution are a necessary (although small) component. However, these topics are comparatively well understood and there are many tools available, so in the absence of good configuration systems, such tools are often extended and pressed into service as configuration solutions.

This section looks at the possible approaches to file distribution and package management and their relationship to the configuration problem.

4.1 File Distribution Tools and Configuration

Once a configuration tool (or a manual process) has determined which files should be installed on a particular node, a file distribution tool is required to copy the necessary files from the repository onto the client during the installation phase. If the requirements then change, the tool may be required to install additional files or to remove existing ones. The comparative simplicity of this process means that it is relatively well understood, and there are a wide range of tools available, many of which provide effective solutions. A relatively small number of files (the "configuration files") will be modified (or generated) by the configuration tool (or manually) to complete the configuration process.

If the diversity and change rate for a group of machines can be minimized, the file selection and modification processes become very simple, and the core of the configuration problem can be reduced to file distribution, perhaps augmented by some simple scripting to customize the resulting configurations for each node. Since file distribution tools are simple and widely available (unlike more complete configuration tools), this can be an attractive approach to managing configurations for small sites or clusters; the cluster nodes or clients are kept as nearly identical as possible and are managed by file distribution and customization scripts. A smaller number of servers will typically have their configurations managed individually (possibly by hand), to avoid the complication of diversity. This is a very pragmatic solution for small sites, and some variation of this is probably the most common approach to configuration management today.

Many "configuration tools" are essentially file distribution tools that have been augmented with additional scripting features in an attempt to provide solutions based on the above approach. In many cases, these can be good, pragmatic solutions for small sites; however, they address only the very lowest levels of the configuration problem, and they are often subject to all the dangers of arbitrary scripting described in section 2.3.

4.2 Package Management

The files to be installed on a particular system can typically be specified in three different ways:

- As a complete disk image (or "opaque" file list).
- As a list of individual files.
- As a list of *packages*, each of which is represented by a list of files and some additional meta-information.

4.2.1 Image-Based Systems

Dealing with complete disk images is often adequate where there is a high degree of uniformity in a system (e.g., in a large compute cluster). A golden copy can be created by hand, and this can be installed efficiently on many different hosts—by multicasting, for example—before making minor host-specific customizations. This is a very coarse granularity, however, and the configuration tool has no "knowledge" of the details of the configuration; this makes such tools unsuitable for very diverse environments, and "higher-level" issues, such as management of inter-machine relationships, are not normally supported (and are likely to be difficult to integrate). SystemImager [48] is a typical (and effective) tool, based on image copying.

4.2.2 File-Based Systems

At the opposite extreme, some tools deal with simple lists of files to be installed. Usually, the file lists can be constructed per-machine, and this provides a simple way of supporting very diverse environments—simple inclusion directives in the file lists are usually sufficient to structure the file sets for different classes of machines. The flexibility of this approach and the scope for diversity usually make it preferable to an imaging approach, except where the machines being configured are guaranteed to remain very homogeneous and the efficiency gains of imaging are desirable (and proven). Radmind [12] is a typical tool based on file copying.

4.2.3 Package-Based Systems

The file sets for most modern operating systems are managed as collections of packages. Each package consists of a set of files, together with a variety of meta-information. Depending on the packaging system, this may support:

- *Clean addition/removal*—Each file installed by the package is recorded, and packages can be removed easily, leaving no significant trace of their previous installation.
- *Dependency management*—Each package can specify dependencies so that it is impossible to install an application without its prerequisites, or to

install an application which conflicts with an existing one.

- *Verification/repair*—The checksums of the package files can be recorded, so that unwanted modifications can be detected and corrupt files reinstated.
- *Remote transport*—Packages can be automatically retrieved from a remote repository.
- *Script execution*—Scripts can be executed after package installation or removal, to perform some customization, although this can be a problem as well as a benefit (see section 4.3, below).

The first two properties above are particularly important—without these, reliable control over the installed packages is impossible.

Most operating systems have a preferred package management tool which is normally used for the distribution of the OS itself: RPM [28] (RedHat Linux), Pkg [13] (Solaris), MSI [57] (Windows), etc. Usually, most software for a particular operating system is available in the vendor's preferred package format.[1] Despite some attempts to develop a standard (Posix P1003.7.2), there is currently no ubiquitous cross-platform solution, and tools that attempt to address this (e.g., the Grid Packaging Tool [6]) often introduce additional complexity by requiring multiple package management technologies on each machine.

In general, the core software for a machine is best managed using the native package format for the particular platform; most software will already be packaged appropriately, and there will be a range of useful tools available. The granularity of a package is also more natural than a file or a disk image when specifying the required software. In a few cases, there may be a good justification for using imaging, or file copying, but these reasons should be considered carefully (Chierichi et al.'s paper [37] shows one example of a comparative performance analysis). Early solutions based on identifying package files by directory (e.g., Depot [55]) or ownership (e.g., lfu [19]) tend to have been largely superseded by modern package management tools.

4.3 Package Configuration

There are two levels at which configuration information is required by a package management tool:

1. To select the required packages (or files) for each machine.
2. Having installed a package (or file set), to modify some of the installed files (or registry entries, in the case of Microsoft Windows) to reflect the necessary configuration.

Normally, the package selection occurs outside of the package management tool, and some other tool can easily be used to generate and manage the necessary package lists for each node. However, the configuration of the package itself (on the client) is

1. Unfortunately, OS X (Apple) is an exception, and third-party software often uses a variety of different technologies.

often handled by the package management tool; technically, this is a separate function which should be handled by a configuration tool, but the package management tools are normally intended to operate alone, and they often include some post-install configuration ability. This introduces a number of difficulties:

- Configuration of the package only occurs at install time; there is no opportunity to reconfigure the package later to track changing requirements (without reinstalling).
- The configuration process might be designed for manual interaction and may be difficult to interface with a tool that computes the necessary configurations automatically.
- The package management tool may typically want to execute arbitrary code (supplied with the package) as a privileged user, and this may be not be acceptable (e.g., RPM pre/post scripts).
- Configuration scripts included with packages are extremely difficult to write in a sufficiently generic way; for example, a package may need to add an extra user to the system, and this would involve reconfiguration of some other subsystem, which is well outside the scope of the package. Simply adding an arbitrary user (and user ID) to the local password file is unlikely to be appropriate in most cases.

In an ideal situation, the package configuration (as well as the selection) would be under the control of some configuration tool, so that configuration files belonging to the package would be updated appropriately when the configuration was changed, not just when the package was installed. Dependencies in configuration information—such as the requirement for a package to have a specific user ID—would be resolved at a sufficiently high level that the user ID would be created automatically in the appropriate way, even if this implied changes to the configuration of some other node (e.g., an NIS server).[2] Since the information used by different package management tools is very similar (a list of packages and versions), the configuration tool should also be able to present a consistent interface for package management across multiple platforms.[3]

One further problem with most current package management systems is the use of the package name as a unique identifier for the package contents. RPM, for example, usually encodes a (hopefully unique) package name, the target architecture, and versioning information into the filename of the package. Most tools then assume that packages with the same filename have the same contents. However, the contents of packages can vary in other ways; for example, the code may have been linked to different libraries (static or dynamic), and it is usually difficult for a configuration tool to distinguish between these.

2. This is an example of the relationship management problem discussed in the previous chapter.

3. LCFG components provide a consistent interface to both RedHat Linux and Solaris package management.

4.4 Some Other Considerations

It is certainly important for a package management system to present a declarative interface to the higher-level configuration system; that is, the configuration system should specify the packages required on the machine, and the tool should translate this into the necessary addition and removal operations. As with other configuration parameters, this declarative approach allows the tool to continuously monitor the actual state of the system and add or remove packages as necessary, to match the requirements. If this is not the case, it is very easy for the system to get out of step with the specification, caused either by users manually adding or removing packages or by the machine missing some update operation. Support for declarative package lists often requires an additional layer—for example, updaterpms (see the LCFG guide [18]) is one tool that provides this functionality on top of the basic RPM mechanism. Tools such as rsync perform the equivalent function at the file level.

The transport protocols supported by the tool may also be an important issue. Some tools have no transport of their own, some require an underlying remote file system, and others support a range of protocols, including custom protocols and standards such as ssh and http. Multicast protocols can be very useful when installing complete clusters or laboratories, and tools such as SMS [10] provide bandwidth throttling to allow background package distribution that does not impair the foreground network performance.

4.5 Some Key Points

- The bulk of the files on any system can be installed by package management software. This provides a number of benefits over disk-level or file-level management.
- Configuration involves much more than file distribution and package management; the configuration of the system determines which packages are to be installed and the contents of the configuration files for each package.
- In simple cases, if the amount of diversity between machines can be minimized, configuration can be handled using a package management tool, together with some ad hoc process for customizing the configuration files. This is a common technique in the absence of more sophisticated configuration tools.
- This simple solution becomes unworkable as the complexity increases, and it does not address higher-level configuration issues.
- More sophisticated configuration tools which can address these issues need to rely on a solid package management and distribution framework.

5. Some Sample Tools

As was noted in the previous chapter, package management and distribution are comparatively well understood; although there are a wide range of tools available, it is relatively easy to outline the different approaches and place the tools in broad categories. In contrast, there are currently no tools that address the full scope of the configuration problem as stated in chapter 1; those that do attempt to solve a significant part of the problem emphasize different issues, use different terminology, and are very difficult to compare.

This chapter presents a selection of tools from this category; this is not intended to be comprehensive, and there will certainly be important and useful tools that are not even mentioned. The aim is to compare a number of tools that emphasize different approaches and, by using common terminology and criteria, to provide a context in which to evaluate other tools. Following a brief paragraph of description and background, each tool is presented under the following headings:

1. *How Does It Work?*—A brief explanation of how the tool works.
2. *Some Observations*—Some important observations (mostly limitations) about the tool.
3. *In Conclusion*—Some conclusions about the role of the tool, especially considering in which situations it is likely to be appropriate.

5.1 Cfengine

Cfengine is probably one of the most widely used, and earliest, tools to address the system configuration problem, with a USENIX paper on it dating from 1995 [32]; it is now a mature product, freely available on most UNIX systems. Cfengine includes a rich set of features which are well-documented and discussed elsewhere—the cfengine Web site [31] includes pointers to a range of material. This section presents a very brief outline of cfengine's design and discusses the relationship with the various aspects of the configuration problem outlined in chapter 1. Readers unfamiliar with cfengine will probably want to read some of the cfengine references first: Æleen Frisch's article [49], for example, provides a good overview of the system.

5.1.1 How Does It Work?

The cfengine Web page summarizes the tool as follows:

Cfengine, or the configuration engine, is an autonomous agent and a mid-

> dle to high level policy language and agent for building expert systems to administrate and configure large computer networks.
>
> Cfengine is designed to be a part of a computer immune system. It is ideal for cluster management and has been adopted for use all over the world in small and huge organizations alike. [31]

The current version of cfengine is capable of a number of additional functions, such as anomaly detection, and a practical implementation requires some attention to various details, such as secure transport. However, the core of the cfengine functionality can be summarized as follows:

- In a typical installation, each host runs a client program (cfagent), which interprets a file of declarations (called a "script") describing desirable system states. The scripts may be downloaded automatically from a central server running cfservd, but they may be (and often are) distributed in other ways.
- Cfengine selects matching declarations for each host by comparing various characteristics of the host against "guard phrases" on the declarations. This allows a common script to be shared by many hosts, and the active subset of declarations is determined by various conditions such as the architecture, the system type, or the result of user-defined probes.
- Cfengine examines the active declarations and compares them with the actual state of the machine. If there is a mismatch, cfengine takes an appropriate action to modify the machine state and bring it into line with the declaration.
- Cfengine may be run on demand, or automatically at regular intervals.

There is a range of built-in, low-level primitives for the actions, including operations such as file editing, process management, and (arbitrary) script execution. These are normally idempotent, and cfengine reorders them before execution; hence the scripts are usually considered to be declarative, although the action order is often significant in practice.

The concept of *convergence* is central to the cfengine philosophy; the scripts embody some idea of the designated configuration, and when cfengine detects differences between this and the actual configuration, some action will be taken to move the actual configuration towards the desired one.[1] It is recognized that this process can never be absolutely certain, and configurations may take several passes to stabilize; it is even possible for configurations to oscillate or otherwise fail to converge (although this is usually considered an error). This process is described as self-healing or computer *immunology* [33] and is particularly useful in a non-proscriptive environment, where accidental configuration errors, introduced by manual intervention, can be automatically "healed."

1. No action is taken unless it is required.

5.1.2 Some Observations

Cfengine has no preconditions for use; it can be applied to an existing system in any state to enforce some specific part of the configuration in a declarative way. This makes it a natural successor to the use of shell scripts for ad hoc configuration.

Although cfengine is described as a “high-level” tool, this term is relative, and it is certainly not aimed at the higher-level issues identified in figure 2.1. Scripts tend to manipulate files, permissions, and processes rather than services and node relationships. The lack of constructs for creating parameterized modules has also prevented the effective development of higher-level shareable libraries, and most sites appear to define their own procedures from scratch, usually operating in different ways. In theory, it should be possible to create compilers that translate higher-level requirements into cfengine scripts, and this is the kind of approach taken by tools such as SysNav [14, 62]. However, cfengine itself has no natural way of expressing relationships between objects, even if those objects exist on the same node.

Cfengine operates essentially at a “host” level, and although it is possible to specify configurations for whole sets of hosts, it is not easy to specify the relationships between those hosts. For example, automatically relating the configuration of some server to the corresponding configurations of the clients is not at all straightforward (see the discussion of relationships in section 1.3). Although cfengine performs self-healing for an individual node, it is difficult to achieve the same autonomic effect at an inter-node service level—for example, to automatically reconfigure a replacement Web server (with all the associated relationships) when the Web server dies. In a cluster situation or a teaching laboratory, cfengine handles the scale (and even the diversity) reasonably well; where the relationships are more complex (such as in a Grid service), cfengine is less obviously useful.

Cfengine is non-proscriptive—at least, it is almost always used in a non-proscriptive way. Most cfengine installations do not control every aspect of every machine’s configuration—other tools and manual procedures are involved. This is particularly true for those hosts with more complex configurations (“servers”). This contributes significantly to cfengine’s popularity; it is possible to take an incremental approach to both the learning and the implementation of the tool. This ease of adoption is clearly an advantage in terms of usability; however, non-proscriptive solutions can lead to serious divergence of configurations (see [42, 40]), and most sites will eventually want complete control over their configurations (certainly, this is essential for any autonomic behavior at a whole-site level). This is possible to achieve with cfengine, but it requires considerable care and discipline.

Cfengine does not provide any explicit support for devolved aspects. The ability to separate concerns will depend very much on the structuring of the cfengine scripts themselves, and it is unlikely that many cfengine-based sites have large numbers of people collaborating on different aspects of the same set of machines.

5.1.3 In Conclusion . . .

Despite the above limitations, cfengine is accessible, reliable, and well known; this makes it a very popular choice for many sites that need something better than completely ad hoc manual configuration management. However, it is not well suited to sites that require a high level of automation or autonomics; cfengine is likely to be difficult to use in these situations.

5.2 LCFG

LCFG was initially developed around the same time as cfengine, with the first publication in 1994 [20]. Like cfengine, LCFG has acted as a testbed for research into system configuration as well as providing a production service (in this case, for over 1,000 very diverse machines in a university computer science department). For reasons that will become apparent, LCFG is more difficult to adopt than cfengine and less portable; it has therefore tended to be used only by sites which can make a heavy initial investment and can benefit from LCFG's particular strengths. For example, LCFG was initially used by the European DataGrid, which later developed its own tool (quattor [11]) based on the same architecture. This section concentrates on LCFG's approach to the general problems outlined in chapter 1. References to LCFG papers and to the code itself are available from the Web site [8]. The Atlanta paper [24] provides a reasonable overview of the current version.

5.2.1 How Does It Work?

The LCFG core consists of two main parts (see figure 5.1):

- An LCFG client (rdxprof) runs on each managed host. This receives a notification from the server whenever a new configuration is available, and it fetches a single XML document, describing the complete configuration of the host, using a simple HTTP(S) protocol. The profile specifies which *components* should be active on the host and includes a simple set of declarative *resources* (parameters) for each active component. Components are notified by the client whenever their resources change, so that they have an opportunity to reconfigure as appropriate. This may involve regenerating configuration files based on the new parameters, stopping or starting daemons, or performing any other necessary operation.
- The LCFG server (mkxprof) maintains a complete description of the configuration of the entire site, but not by simply listing the parameters for each node; the configuration consists of various aspects (e.g., Web server, Dell GX150), which are usually maintained by different people. The compiler composes the appropriate aspects for each host (resolving conflicts where possible) to generate an XML profile that explicitly defines the value of each resource. The LCFG server monitors the source files; when a change occurs, all the affected profiles are regenerated and the

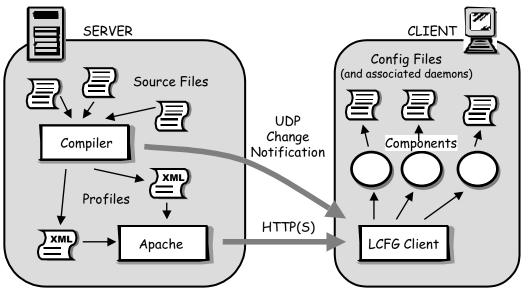


Figure 5.1: The LCFG architecture

appropriate clients are notified. Figure 1.3 shows a sample profile. The `#include` statements reference common aspects, and the other statements set (or override) individual resource values.

The LCFG core also supports bare-metal host installation (obtaining parameters such as disk partitioning information from the profile), package list management, and simple monitoring (the client sends regular status information to the server). Otherwise, the functionality of LCFG is largely determined by the selected set of client components, and these may be part of the core distribution, unofficially contributed, or locally created. The standard component interface, together with templates and other support for component creation,[2] makes sharing of components relatively straightforward.

LCFG has a different philosophy and somewhat different motivations from cfengine. For example:

- LCFG is intended to provide proscriptive configuration management;[3] it is designed for complete automated control of all machine types, from laptops to servers.
- LCFG is intended to provide a simple, declarative interface to the configuration of a whole site that is suitable for use by other tools. This allows configurations to be easily generated, manipulated, and validated by higher-level tools, while being instantiated and maintained by LCFG. It also allows relationships between hosts (related client-server parameters) to be managed automatically.
- LCFG expects configurations to be managed as the composition of requirements from many independent people. These people will have different skill levels and may contribute in correspondingly different ways; a technician may create an LCFG profile, a system administrator may create a particular aspect, and a programmer may create a new component to manage a completely new subsystem.

2. These would be in Perl or shell script.
3. It can, however, be used for non-proscriptive management.

- The use of explicit components, responsible for disjoint areas of the system configuration, creates natural closures (see section 2.3.2) which prevent changes to one aspect of the configuration having an adverse effect on some unrelated aspect.

These properties support the implementation of a sitewide autonomic capability; a high-level tool can interface with LCFG and automatically manage significant reconfiguration of inter-node services (to deal with hardware failures or load-balancing). See [23] for an example of this.

5.2.2 Some Observations

In general, the learning curve for LCFG is considered to be rather high. Components are available for managing many standard subsystems (e.g., automounter, Sendmail, Kerberos), but configuring these requires a knowledge of their supported resources, which may often be somewhat different from the contents of the familiar configuration files (it is often at a higher level). If the existing components do not provide some required functionality, it becomes necessary to write a new component (or modify an existing one), which involves a knowledge of the component interface as well as basic scripting skills. The profile syntax for LCFG has also evolved over several years and is widely recognized as awkward and confusing.

The central compilation of the profiles is necessary to enable inter-machine relationships to be computed (the so-called *spanning maps*). This can be a bottleneck for large sites, however, taking several seconds per machine (30 to 40 minutes for a change affecting 1000 machines). As well as being inconvenient, this is a significant barrier to those autonomic applications that require a rapid reconfiguration response.

The LCFG components usually encapsulate some knowledge of the subsystem they are intended to configure.[4] For example, the Sendmail component understands a relay resource which ultimately defines the corresponding parameter in the sendmail.cf file—by abstracting this resource (rather than just specifying the parameters for sendmail.cf), we have the possibility of using different components (and even different mail agents) with the same configuration specifications. However, this does mean that components are rarely portable between different operating systems without some work.

Small changes to some aspect on an LCFG server can initiate massive reconfiguration of an entire network. By default, hosts will reconfigure as soon as the server has regenerated their profile and dispatched the notification. For most reconfigurations of related servers, the reconfiguration order is important; for example, we would like to ensure that a new NFS server is available before transferring clients to it (and before removing the old server).

5.2.3 In Conclusion . . .

LCFG is designed to perform high-level, proscriptive configuration management,

4. This is not true of all components—the file component, for example, can be used to generate arbitrary files and is highly portable.

including the management of relationships between machines. This provides a good basis for tightly controlled configurations and high-level autonomics in a diverse environment. However, the learning curve and the level of commitment are high, and the portability is low.

5.3 Microsoft Tools

Microsoft "Active Directory" [1] was introduced with Windows 2000 and includes a number of technologies, such as Kerberos (authentication) and LDAP (directory services) which have been integrated into a standard framework for system configuration on the Microsoft Windows platform. SMS (System Management Server) is a separate product capable of running independently of Active Directory and containing some additional functionality (monitoring and "inventory control"), as well as an alternative approach to software delivery. Active Directory is now the de facto standard for Microsoft sites, being simple for small sites to use and sufficiently extensible for customized development in larger organizations. This section looks (only) at the configuration-related aspects of these products.

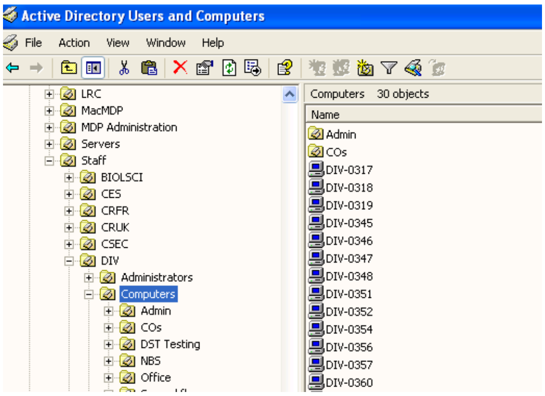
5.3.1 How Does It Work?

The configuration parameters for different aspects of a machine are stored in Group Policy Objects (*GPO*s) in the Active Directory (LDAP) and in shared file space. These are attached to various nodes in the LDAP hierarchy corresponding to either the machine or the user (see figure 5.2). The per-user and per-machine settings are normally disjoint,[5] but this provides a uniform way of handling both. An individual machine downloads all the GPOs on the appropriate branches, applying them so that values in more specific policies override corresponding values in more general ones.

The resulting values for the configuration parameters are made available to modules on the client, known as "client-side extensions," which can take any arbitrary action to implement the necessary configuration settings (similar to LCFG components). The client-side extensions are called on reboot or at regular (but infrequent) intervals; they are not, however, called automatically when the profile changes.[6] Each client-side extension also includes a server-side module to provide GUI-based editing of its own parameters in the GPOs. Custom extensions are difficult to create and are probably beyond the scope of the average system administrator. The closed source also prevents existing extensions from simply being extended or used as templates for new ones. Microsoft itself provides comparatively few extensions, preferring to rely on simpler approaches to customization, such as an extension that provides "administrative templates" for setting arbitrary registry values.

5. The per-user settings are only applied on login to the user at the console (not remote connections), and it is up to individual applications to resolve any conflicts with the per-machine settings.

6. In fact, this would be difficult, since the server does not necessarily know which clients are affected by a change.



```
LRC
MacMDP
MDP Administration
Servers
Staff
  BIOLSCI
  CES
  CRFR
  CRUK
  CSEC
  DIV
    Administrators
    Computers
      Admin
      COs
      DST Testing
      NBS
      Office
```

```
Computers   30 objects
Name
Admin
COs
DIV-0317
DIV-0318
DIV-0319
DIV-0345
DIV-0346
DIV-0347
DIV-0348
DIV-0351
DIV-0352
DIV-0354
DIV-0356
DIV-0357
DIV-0360
```

Figure 5.2: The Active Directory hierarchy

The standard package format for Microsoft package management is *MSI*; this includes the ability to execute arbitrary scripts at pre/post install time but does not explicitly support dependency information. Packages specified in the Active Directory for a particular machine are automatically downloaded and installed, usually at the next boot or login, although they can be installed on-demand the first time an attempt is made to run them. SMS provides an alternative software installation mechanism which has some different features, such as the ability to throttle the network load, and support for different transport mechanisms and package formats.

Microsoft's standard bare-metal installation tool has some limitations, and third-party tools are often used, even if the machine is subsequently managed using Active Directory.

5.3.2 Some Observations

The configuration classes of machines rarely fall into a strict hierarchy; they are determined by a range of intersecting aspects, including the hardware type, the attached peripherals, and the physical location. This makes the predominance of a single (LDAP) hierarchy in the configuration classes inappropriate. By using access control lists to block access to certain GPOs from machines in certain groups, it is possible to perform more complex aspect composition, but this is an unnatural and confusing process.

The Microsoft configuration tools appear to address the obvious need for configuring large numbers of client machines. In practice, they do not appear to be widely used for (proscriptive) server management, and it is not clear that this is currently practical. The tools do not address the higher-level issues of configuring relationships between nodes at a service level, and while the GUI interface (see figure 5.3) is adequate for setting individual configuration parameters, a different approach is probably necessary to support higher-level specifications.

In practice, there still appear to be a number of difficulties with package management; vendors have not yet completely embraced the MSI technology, and it is com-

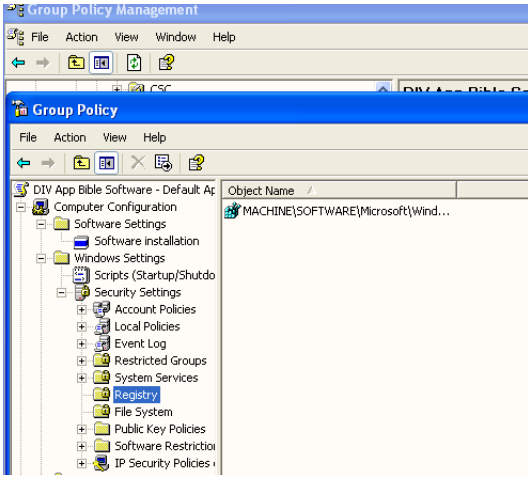


Figure 5.3: An Active Directory group policy

mon to find packages that have their own install technologies and/or some sort of problem during installation. Creation of MSI packages is not encouraged by the difficult learning curve and the need for expensive tools.

5.3.3 In Conclusion . . .

Active Directory provides the de facto standard for configuration management of Microsoft Windows, and good reasons would be needed to justify diverging from this. However, Active Directory is clearly targeted at desktop management and does not attempt to address the more complex configuration requirements of whole-site configuration or autonomics.

5.4 CDDLM, CDL, and SmartFrog

All of the technologies discussed so far have evolved via a bottom-up approach to the configuration problem; they provide immediately useful, practical solutions to many low-level problems, but they do not provide a clear route to addressing many of the higher-level concerns. The technologies discussed in this section are interesting because they have been explicitly developed in response to the needs of highly distributed, autonomic Grid services. This provides a useful example of how some higher-level issues—for example, the problems of change sequencing (see section 3.3) or the concern with managing inter-organizational resources—might be approached. See reference [26] for a discussion of this work with regard to more traditional system administration tools.

The CDDLM working group [5] of the Global Grid Forum (GGF) is concerned with developing standards for "configuration, deployment and lifecycle management of Grid services." This is motivated by the need to "deploy, configure and manage large distributed Grid applications"; however, this shares many problems with system configuration, and the results are extremely relevant. In particular, *CDL* (Configuration Description Language) [63] is a viable candidate for a standard way of representing system configuration information. SmartFrog (Smart Framework for Object Groups) [50] is an object-oriented framework for managing distributed components which can be configured using a declarative configuration language. The SmartFrog language[7] provided much of the inspiration for CDL, and the scope and semantics are very similar (but SmartFrog has a more human-friendly syntax).

5.4.1 How Does It Work?

As its name suggests, the CDDLM working group covers three areas:

- A language for describing declarative attributes of configurable components.
- Mechanisms for deploying components on remote machines.
- Management of the "lifecycle" (completion, failure, etc.) of these components.

The language aspects of CDDLM (CDL) are particularly interesting because the features are comparable to other system configuration languages. Although the semantics of CDL are similar to those of SmartFrog, CDL is XML-based; this makes parsing and interoperability very easy but authoring and readability difficult (presumably, tools will be developed to alleviate this). The CDL model consists of a hierarchy of components, each of which has a set of configurable attributes. The model does not cover the semantics of the components themselves, or even define the allowed components or properties—as already noted, this level of definition appears to be a good practical compromise. A familiar template mechanism provides value-inheritance. For example:

```
<WebServer>
     <hostname>www.example.com</hostname>
     <port>80</port>
     . . .
     </WebServer>
<AppServer cdl:extends="WebServer>
     <hostname>apps.example.com</hostname>
</AppServer>
```

Of particular interest is the reference mechanism, which allows *lazy references* whose values are substituted dynamically at the time the component is deployed. This provides a more dynamic way of establishing relationships between distributed components than the LCFG *spanning maps*, but the implementation involves complex runtime communication.

The deployment aspects of CDDLM are also interesting because they address the problem of sequencing the deployment of a complex distributed service in a reliable and orderly manner. However . . .

7. SmartFrog was developed by Hewlett-Packard but is now open sourced.

5.4.2 Some Observations

The CDDLM "lifecycle" model works well for many application-level services, the classic example being a multi-tiered Web service that includes front-end Web servers and back-end database servers, all of which must be deployed in the right order and connected by relating parameters. However, this approach is less obviously suited to many lower-level infrastructure services: minor changes to the configuration of a Kerberos service, for example, would involve undeploying the service and redeploying a new one; an approach based on adaptive reconfiguration of running services would seem more suitable.

The language aspects of CDDLM are more relevant to system configuration and could well point the way towards some commonality in languages for configuration specification. It is unfortunate that the standard mixes both basic property specifications (which could well be standardized) and higher-level operations such as templating and references (on which there is less agreement).

5.4.3 In Conclusion . . .

These technologies form a comparatively abstract framework, and they are unlikely to make a practical basis for management of site infrastructure in the near future. However, they do illustrate a higher-level approach to system configuration which is likely to become more prevalent, and system administrators may encounter them in practice as a way of managing specific (Web) services at their sites.

5.5 Other Tools

There are numerous other tools that could be classified in some way as "configuration tools," and many of these have some interesting features or niche applications. Unfortunately, there are also many home-grown tools that have little value outside the specific application to their originating organization. Most projects considering configuration issues will want to perform their own surveys of existing tools, although published surveys from other projects are a useful starting point; see, for example, references [29, 21], as well as recent LISA conference proceedings.

BCFG [44] is one example of a more recent tool. This has a tighter coupling between the server and the clients.

5.6 Some Key Points

- Most practical configuration tools have evolved over time from the bottom up and are not well suited to higher levels of automation.
- Tools and standards with the potential to address these issues are not mature enough or easy enough to use, for large-scale practical deployment.
- Traditionally, tools have largely been developed in isolation, with no common standards or interoperability.

6. Theory, Research, and Current Issues

System configuration theory is currently at a stage comparable to the early days of programming in assembly code; most sites are managed using their own ad-hoc techniques and a mixture of tools. Most of these tools and the procedures for their use have been developed, often as pragmatic solutions by the administrators themselves, without a good understanding of the fundamentals. Computer programming has developed into a discipline capable of supporting reliable, large-scale applications by basic research into fundamentals such as language semantics and ways of automatically generating low-level programs from specifications that are closer to the requirements (high-level languages). Addressing the challenges of higher-level configuration will require comparable developments in configuration theory. The effort and the skills required for this research are not within the scope of the average system administrator, but most theoreticians, vendors, and grant-awarding bodies have not yet been fully convinced that this is a coherent area worthy of their attention. There are also comparatively few people who have both a good understanding of the practical problems and the time and skills to develop the theories necessary for their solution. However, despite the lack of a clear theoretical core, some interesting work is starting to appear from various directions, and this is beginning to have an impact on practical understanding and on the development of new tools. This chapter presents an overview of some of these directions.

6.1 Fundamental Theory and Models

Modern programming techniques are based on sound theoretical models. There is general agreement about the meaning of terms such as "program" and the theoretical framework in which to discuss different languages and their semantics (see section 6.2). There is no such agreement about the meaning of "configuration" or "configuration language" in relation to system configuration. It is not appropriate to discuss these issues in detail here, but the following example should give a flavor of the possibilities and show how theory affects some of the practical issues discussed in earlier chapters.

One approach to formalizing the notion of a configuration tool is to consider the state of the system disk to represent the configuration of a machine (in practice, there are other issues, such as the physical connections between the machines, but this is a reasonable simplification). It is then possible to ask, "When should two configurations be considered equivalent?" Clearly, a large proportion of the possible disk states will be completely random and useless, and it will often be helpful to consider these as equivalent (broken). In other cases, there will be a large number of states which are effectively indistinguishable (in behavioral terms), and it may be useful to consider them as equivalent (perhaps they vary only in the arrangement of blocks in the disk free list, for example). This leads to the notion of configurations being equivalence classes (in a mathematical sense) on the sets of possible disk states. The granularity of these equivalence classes is one way of characterizing a configuration tool; for example, this notion formalizes the difference between the extreme approaches to configuration specification described in section 1.2.

Given the above definition of a configuration state, it is then possible to define a configuration tool as a function[1] that transforms a machine into a state that satisfies certain desired properties, i.e., the configuration specification. This simple definition immediately leads to a whole range of issues. For example:

- If the equivalence classes are well defined, the same configuration transformation should always take machines that start in the same state to the same final state. This is not always the case for current practical tools, which expose hidden *preconditions.*
- There will be some states from which the tool cannot recover (e.g., if it deletes itself from the disk).
- If there is a choice of final states that satisfy the requirements, how should the actual state be chosen?

It is surprising how quickly a small amount of such theory can shed light on the root causes of real practical problems.

6.2 Languages and Semantics

There is common agreement that special configuration languages are necessary and that these are very different from programming languages. Such languages need to provide support for those features that have been shown to be important, such as the idempotence and closures discussed in section 2.3. Other issues include the ability to specify configuration aspects in a more natural way, avoiding conflicts, and the ability to specify loose constraints to support autonomics (see, e.g., [52]). These issues are less well understood.

The *semantics* of a language defines the meaning of language statements, while the *syntax* defines the arrangement of symbols used to write the statements. For example,

1. The configuration transform is technically often not a mathematical function, since it usually is defined only on part of the domain and often has multiple possible outcomes.

the following statements have a different syntax but (assuming a suitable language definition) identical semantics:

```
(plus 2 23)
0x2 + 0x17;
2 23 +
ld #2 ; add #23
```

As long as there is a clear formal model of the underlying system, it is possible to define the semantics of a language in a formal way, as a mapping between the symbolic statements of the language and the effect that each statement has on the model. Most modern programming languages have a written definition of their semantics (in a more or less formal notation), and this has the following advantages:

- It is theoretically possible to prove that programs have the desired effect. More practically, it is usually easier to convince oneself that an operation will have the intended outcome if the semantics are clear and well understood.
- Uncertainties and unexpected behavior of the language are highlighted.
- Multiple implementations of the language are more likely to behave in the same way.

In contrast, most current configuration languages have very informal semantics; there are statements whose meaning is not obvious, and the ultimate definition of the language semantics is the code of the (usually one) implementation. The CDL (see section 5.4) is an example of an attempt to define a clear semantics for a configuration language, and to encourage multiple implementations.

6.3 Distributed Deployment and Reasoning

In an ideal world, it would be possible to gather all of the requirements for the configuration of a complete site into a central location, use this to compute the detailed configurations of all the individual nodes, and then deploy the computed configurations onto the nodes. In practice, this approach is becoming less tenable:

- Both nodes and the connections between them are unreliable.
- There is a significant latency in both computation and deployment.
- Parts of the network (and aspects of individual nodes) may be owned and managed by different people or organizations.
- The requirements originate from disparate sources. Both human and automatic input, from various parts of the network, may generate constraints.

Harnessing more decentralized (peer-to-peer) techniques to solve these problems is not straightforward. Not only does the deployment of the configurations have to be distributed, but so, too, do the reasoning and computation of the configuration details themselves. For example, a group of machines might use some distributed technique to agree among themselves on which machine should act as a print server (see [30]). This information then needs to be communicated to the potential clients, perhaps by modifying the DNS or perhaps by using some protocol such as Zeroconf [16]. If this print server subsequently fails, the group will need to agree on a replacement in a decentralized way, and all affected services will need to be reconfigured appropriately. Performing general reasoning in this way in a highly distributed and unreliable environment is very hard, but this is necessary for more complex, interdependent services, such as a multi-tiered Web service. Although the decisions about the details of the configuration may be computed in a distributed way, they still need to conform to some centrally agreed high-level policy; see, for example, reference [36].

"Promise Theory" [35] is a new approach to coordinating behavior in a distributed environment, which is expected to form the basis for the next version of cfengine. This is a graphical technique for combining simple "promises" or assertions of constrained behavior; a promise is given by an autonomous agent who cannot be forced to do anything it does not wish. Agents may, however, promise to comply with other agents' wishes and, in this way, build up familiar structures such as dependencies.

These problems are clearly in the realm of research, and most practical configuration systems currently perform any necessary coordination in a centralized way (apart from the use of protocols such as Zeroconf to handle very specific applications).

6.4 Configuration Synthesis

Supporting high-level configuration specifications requires systems that can perform automated reasoning [59]. For example, translating the statement "I need two DHCP servers on each subnet" into the corresponding low-level parameters is not a straightforward compilation process. There may be several possible solutions, and the choice of an appropriate one is likely to depend on other constraints, either provided explicitly or implied by the environment. Sanjai Narain provides a good practical example in reference [58], which describes the automatic configuration of a fault-tolerant VPN. This is a very realistic example, and it is sufficiently complex that the automated solution highlights problems that are not immediately obvious.

Although it is possible to use general-purpose logic languages such as Prolog for computing specific configurations from general requirements, there are special-purpose constraint solvers that are more appropriate (such as the one used by Narain). Even with these tools, however, performance is an issue; it is easy to specify constraints in an impossibly inefficient way. Slight changes to the constraints can also lead to a complete recomputation of the configuration, possibly resulting in wide-ranging changes to the entire site. Of course, performing constraint solution in a distributed environment (as was described in the previous section) is even more complex (although, in theory, it does offer the potential for distributing the computation).

For practical use, automated reasoning systems must provide clear explanations of their actions and have the ability to perform "what-if" analyses. Using such systems in real time requires a good deal of confidence—administrators need to be confident that the corporate Web server will not suddenly be migrated onto their laptop without explanation!

7. Conclusion

"System configuration" is the process of integrating hardware and software into a single "system" that performs a specified overall function and continues to do so even as the system components and the requirements change.

Modern computing sites involve highly complex interactions, and configuration errors are a major and growing source of failures. System configuration tools are currently in a primitive state of development and are incapable of addressing the complete configuration problem. Most sites therefore employ a mixture of automatic and manual processes.

The current immature state of system configuration has led to the development of many different tools with different approaches, fostering ad-hoc, bottom-up solutions. Good system configuration practice involves an awareness of limitations and a careful selection of manual and automatic procedures appropriate both to the resources available and to the operational priorities of the site.

7.1 The Future

The trend toward more complex computing "fabrics" is producing increasingly complex interactions, both within and between sites. The correctness and agility that this demands from the configuration system require a level of automation beyond the currently available solutions.

For the practicing system administrator, this is likely to entail a significant shift, moving from low-level concepts (e.g., configuration files) to high-level concepts (e.g., services). This may involve radically different ways of specifying system requirements, replacing certainties with probabilities and absolute values with loose constraints.

A new generation of tools will be necessary to support these developments. Progress in this area is currently rather slow, since it involves adopting new theoretical approaches grounded in the solid, practical problem-solving experiences of large and complex sites. This forms the small but growing topic of system configuration research.

Glossary

Since system configuration is a relatively new subject, there is little established, common vocabulary. Apparently generic terms are used by different authors with specific (and often different) meanings. This glossary provides the definition of some common terms as they are used in this booklet. The definitions of technical terms are intended to be informal.

Actual configuration: The current state of the configuration, as deployed on a running system. Because of latency issues, this often differs from the designated configuration.

Aggregation: The collecting together of configuration parameters from many different nodes to form part of the configuration of some other node. For example, the collation of port numbers from the externally visible servers to form part of the configuration of the firewall.

Aspect: A view of part of an entire configuration which is the concern of one person or function: e.g., "Web service," "security," or "student machines." Frequently, individual configuration parameters will be affected by multiple aspects, and it is the job of the configuration tool to compose them to create a final value.

Asymptotic configuration: In a large and evolving system, the configuration requirements will almost always change faster than they can be deployed on all of the nodes. The term "asymptotic configuration" is used to describe the resulting (continuous but changing) discrepancy between the designated configuration and the actual configuration.

Autonomic: Autonomic computer systems are ones that "maintain and adjust their operation in the face of changing workloads, demands, and external conditions, and in the face of hardware or software failures of innocent or malicious origin." See [54].

CDDLM: A working group of the Global Grid Forum (GGF) concerned with developing standards for "configuration, deployment, and lifecycle management of Grid services."

CDL: A Configuration Description Language developed for the CDDLM.

CIM: Common Information Model. Detailed schema for the configuration information required by various different systems. Developed by the DMTF.

Cloning: Making an identical copy of a system, usually by copying a physical disk image.

Closure: A self-contained set of configuration parameters which do not interact with the parameters from any other closures.

Component: The configuration tool LCFG uses this term to refer to a client-side code module which manages the configuration of one particular subsystem.

Composition: Combining configuration parameters from different aspects to produce a definitive set of configuration parameters. This usually involves resolving conflicts when the aspects have overlapping concerns.

Convergence: The process of modifying the actual configuration of some existing system to bring it closer to the designated configuration.

Declarative description: A specification of some property of a configuration, as opposed to a procedural description of how to manipulate the configuration.

Deployment: The process of configuring a machine (usually remotely) to conform to a particular low-level configuration description.

Designated configuration: The desired state of the configuration of some system. Because of latency issues, this is often different from the actual configuration deployed on the running system.

Devolved management: This occurs when different aspects of the configuration of a site (or, particularly, an individual machine) are under the control of different people.

DMTF: The Distributed Management Task Force, Inc. (DMTF) is the industry organization leading the development of management standards and integration technology for enterprise and Internet environments. DMTF standards provide common management infrastructure components for instrumentation, control, and communication in a platform-independent and technology-neutral way. DMTF technologies include information models (CIM), communication/control protocols (WBEM), and core management services/utilities. See [2].

Fabric: The underlying infrastructure of a computing site (as opposed to the applications).

Federation configuration: A system configuration in which loosely connected people or groups have specific requirements for different aspects of the configuration.

Golden copy: The master copy of a set of files (usually a whole disk image) which is cloned onto a set of machines to produce identical configurations.

GPO: Group Policy Object. A set of Active Directory configuration parameters which can be applied to specific groups of machines.

Grid: "The Grid is a service for sharing computer power and data storage capacity over the Internet." See [7].

High-level configuration: Configuration at a level that specifies services and relationships between machines rather than low-level details, such as file contents and processes on individual machines.

Idempotent: An operation in which multiple applications are equivalent to a single application.

Immunology: The ability of a configuration tool to continuously correct configuration errors in a way analogous to a biological immune system.

Inheritance: In configuration languages, inheritance usually implies value inheritance, i.e., some structure inherits all the values from some other structure, usually with the intention of overriding some of the values with more specific ones. This is different from the type inheritance of most programming languages, where it is the structure (rather than the value) of some other object that is inherited.

Lazy reference: A reference allows one configuration parameter to specify the value as being "the same as" some other value. If the other value is not known until the configuration is deployed, this value cannot be instantiated until deployment time and is therefore known as lazy.

Low-level configuration: Configuration at a level that deals with details such as file names and processes rather than high-level concepts, such as services and relationships between machines.

Model: The model supported by a configuration tool determines the types of objects that can be described and the relationships and operations that can be represented. There is an important distinction between this and the language used to describe the model.

MSI: Microsoft Windows Installer. The standard Microsoft packaging tool.

Package: A collection of files intended to be installed as a unit. The files are usually bundled together with some meta-information, such as dependencies, versioning, and installation scripts.

Preconditions: Some pre-existing condition of the machine state which a configuration tool requires to make a specific configuration change. Hidden preconditions occur when the tool makes non-obvious assumptions about the machine state that are likely to be untrue.

Procedural description: A description of the process for manipulating a configuration, as opposed to a declarative description of the required final state.

Profile: The complete set of configuration parameters for a single machine (an LCFG term).

Proscriptive configuration: Defining the entire set of configuration parameters for a machine, as opposed to defining only a subset and allowing the rest to be defined manually (or by some other tool).

Prototype: A set of configuration parameters intended to be used as a template for several different instances. Usually the individual instances will override some values from the prototype.

Resource: A single configuration parameter (an LCFG term).

Scripting: Modification of a configuration using some procedural process, such as a script or program.

Self-healing: Automatically correcting configuration errors which have been introduced, for example, by system failures or human error.

Semantics: The meaning of the statements in a configuration language; this is in contrast to the syntax of the language, which defines the arrangement of symbols.

Spanning map: A construction in the LCFG language used to aggregate values from a group of machines for incorporation into the configuration of some other machine.

Specialization: The process of overriding some values in a prototype to customize it for use in a specific instance.

Syntax: The form in which a (configuration) language is written. This is distinct from the semantics of the language, which describes the meaning of the various statements.

WBEM: "Web Based Enterprise Management is an industry initiative to provide management of systems, networks, users and applications across multiple vendor environments." See [15].

References

[1] Active Directory overview: http://www.microsoft.com/windows2000/server/evaluation/features/dirlist.asp.

[2] Distributed Management Task Force (DMTF): http://www.dmtf.org/home.

[3] Dynamic reconfiguration of Grid fabrics. The OGSAConfig project: http://groups.inf.ed.ac.uk/ogsaconfig/.

[4] GConf: http://www.gnome.org/projects/gconf/.

[5] The GGF CDDLM working group: https://forge.gridforum.org/projects/cddlm-wg.

[6] The Grid Packaging Tool (GPT): http://www.gridpackagingtools.org/.

[7] GridCafe: The place for everybody to learn about the Grid: http://gridcafe.web.cern.ch/gridcafe/.

[8] LCFG: http://www.lcfg.org/.

[9] LISA: The Large Installation System Administration Conference: http://www.usenix.org/events/byname/lisa.html.

[10] Microsoft system management server: http://www.microsoft.com/smserver/.

[11] Quattor: http://www.quattor.org/.

[12] Radmind. University of Michigan: http://rsug.itd.umich.edu/software/radmind/.

[13] Sun Microsystems: Application packaging developer's guide: http://docs.sun.com/app/docs/doc/806-7008/.

[14] Systems Navigator (SysNav): http://www.sysnav.com/.

[15] What Is WBEM? http://wbemservices.sourceforge.net/#WBEM.

[16] Zeroconf. IETF Zeroconf Working Group: http://www.zeroconf.org/.

[17] Autonomic computing: IBM's perspective on the state of information technology. Technical report, IBM Corporation, 2001. http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf.

[18] P. Anderson. The complete guide to LCFG. Technical report, University of Edinburgh: http://www.lcfg.org/doc/guide.pdf.

[19] P. Anderson. Managing program binaries in a heterogeneous UNIX network. In Proceedings of the 5th Large Installation Systems Administration Conference, pages 1–9, Berkeley, CA, 1991. USENIX: http://homepages.inf.ed.ac.uk/dcspaul/publications/LISA5_Paper.pdf.

[20] P. Anderson. Towards a high-level machine configuration system. In Proceedings of LISA '94: 8th USENIX Systems Administration Conference, pages 19–26, Berkeley, CA, 1994. USENIX: http://www.lcfg.org/doc/LISA8_Paper.pdf.

[21] P. Anderson, G. Beckett, K. Kavoussanakis, G. Mecheneau, and P. Toft. Technologies for large-scale configuration management. Technical report, GridWeaver Project, December 2002: http://www.gridweaver.org/WP1/report1.pdf.

[22] P. Anderson, G. Beckett, K. Kavoussanakis, G. Mecheneau, P. Toft, and J. Paterson. Experiences and challenges of large-scale system configuration. Technical report, GridWeaver Project, March 2003: http://www.gridweaver.org/WP2/report2.pdf.

[23] P. Anderson, P. Goldsack, and J. Paterson. SmartFrog meets LCFG—Autonomous reconfiguration with central policy control. In Proceedings of LISA '03: 17th Large Installation Systems Administration Conference, Berkeley, CA, 2003. USENIX: http://homepages.inf.ed.ac.uk/dcspaul/publications/lisa03.pdf.

[24] P. Anderson and A. Scobie. Large-scale Linux configuration with LCFG. In Proceedings of the 4th Annual Linux Showcase & Conference, Atlanta, pages 363–372, Berkeley, CA, 2000. USENIX: http://www.lcfg.org/doc/ALS2000.pdf.

[25] P. Anderson and E. Smith. Dynamic reconfiguration for Grid fabrics: Case studies. Technical report, OGSAConfig project, June 2004: http://groups.inf.ed.ac.uk /ogsaconfig/papers/report1.pdf.

[26] P. Anderson and E. Smith. System administration and CDDLM. In GGF12 CDDLM Workshop. GGF, 2004: http://homepages.inf.ed.ac.uk/dcspaul /publications/ggf12.pdf.

[27] P. Anderson and E. Smith. Configuration tools: Working together. In Proceedings of LISA '05: 19th Large Installation System Administration Conference, Berkeley, CA, 2005. USENIX: http://www.usenix.org/events/lisa05/tech/anderson.html.

[28] E. C. Bailey. Maximum RPM. RedHat Software Inc., 1997.

[29] M. Barroso. Datagrid: WP4 report on current technology. Technical report IST-2000-25182, 2001. http://hep-proj-grid-fabric.web.cern.ch /hep-proj-grid-fabric/Tools/DataGrid-04-TED-0101-3_0.pdf.

[30] G. Beckett, G. Mecheneau, and J. Paterson. The GPrint Demonstrator. Technical report, GridWeaver Project, December 2002. http://www.gridweaver.org/WP4/report4_1.pdf.

[31] M. Burgess. Cfengine: http://www.cfengine.org/.

[32] M. Burgess. Cfengine: A site configuration engine. USENIX Computing Systems, vol. 8, no. 3, 1995: http://www.iu.hio.no/~mark/papers/paper1.pdf.

[33] M. Burgess. Computer immunology. In Proceedings of LISA '98: 12th Systems Administration Conference, page 283, Berkeley, CA, 1998. USENIX: http://www.usenix.org/publications/library/proceedings/lisa98/full_papers/burgess /burgess.pdf.

[34] M. Burgess. Analytical network and system administration. John Wiley, 2004.

[35] M. Burgess. An approach to understanding policy based on autonomy and voluntary cooperation. Proceedings of DSOM, 2005.

[36] M. Burgess and S. Fagernes. Pervasive computing management (I): A model of network policy with local autonomy. IEEE eTransactions on Network and Service Management (submitted).

[37] A. Chierici, L. dell'Agnello, E. Ferro, and M. Serra. Experience with LCFG installation in DataGrid environment. Technical report, European DataGrid, May 2003: https://edms.cern.ch/file/384844/1/lcfg-scalab-test.pdf.

[38] A. Couch. Why people don't adopt configuration management tools. Presentation. 2004: http://homepages.inf.ed.ac.uk/group/lssconf/config2004/slides /alva/workshop.pdf.

[39] A. Couch. Toward a cost model for system administration. In Proceedings of LISA '05: 19th Large Installation System Administration Conference, Berkeley, CA, 2005. USENIX: http://www.usenix.org/events/lisa05/tech/couch.html.

[40] A. Couch. A (very brief) overview of cfengine. Presentation. 2005: http://homepages.informatics.ed.ac.uk/group/lssconf/config2005e/Slides/cfengine.pdf.

[41] A. Couch and M. Gilfix. It's elementary, dear Watson: Applying logic programming to convergent systems management processes. In Proceedings of LISA '99: 13th Systems Administration Conference, page 123, Berkeley, CA, 1999. USENIX: http://www.usenix.org/events/lisa99/full_papers/couch/couch.pdf.

[42] A. Couch and Y. Sun. On observed reproducibility in network configuration management. Science of Computer Programming, vol. 53, no. 2, pages 215–253, 2004.

[43] E. C. F. Daniel Sabin. Configuration as composite constraint satisfaction. In Proceedings of the (1st) Artificial Intelligence and Manufacturing Research Planning Workshop, G. F. Luger, ed., pages 153–161. AAAI Press, 1996.

[44] N. Desai, R. Bradshaw, R. Evard, and A. Lusk. BCFG: A configuration management tool for heterogeneous environments: ftp://ftp.mcs.anl.gov/pub/bcfg/papers /bcfg-cluster2003.pdf.

[45] N. Desai, R. Bradshaw, S. Matott, S. Bittner, S. Coghlan, R. Evard, C. Lueninghoener, T. Leggett, J.-P. Navarro, G. Rackow, C. Stacey, and T. Stacey. A case study in configuration management tool deployment. In Proceedings of LISA '05: 19th Large Installation System Administration Conference, Berkeley, CA, 2005. USENIX: http://www.usenix.org/events/lisa05/tech/desai.html.

[46] DMTF. Common information model (CIM) specification. Technical report, DMTF, 1999: http://www.dmtf.org/standards/cim/cim_spec_v22.

[47] R. Evard. An analysis of UNIX system configuration. In Proceedings of LISA '97: 11th Systems Administration Conference, page 179, Berkeley, CA, 1997. USENIX: http://www.usenix.org/publications/library/proceedings/lisa97/full_papers /20.evard/20.pdf.

[48] B. E. Finley. VA SystemImager. In Proceedings of the 4th Annual Linux Showcase & Conference, Atlanta, Berkeley, CA, 2000. USENIX: http://www.usenix.org /publications/library/proceedings/als00/2000papers/papers/finley.html.

[49] Æ. Frisch. Top five open source packages for system administrators: http://www.onlamp.com/pub/a/onlamp/2003/05/29/essentialsysadmin.html.

[50] P. Goldsack. SmartFrog: Configuration, ignition and management of distributed applications. Technical report, HP Research Labs. http://www-uk.hpl.hp.com /smartfrog.

[51] M. Holgate and W. Partain. The Arusha project: A framework for collaborative systems administration. In Proceedings of LISA 2001: 15th Systems Administration Conference, Berkeley, CA, 2001. USENIX: http://www.usenix.org/events/lisa2001 /tech/full_papers/holgate/holgate.pdf.

[52] A. Holt and J. Hawkins. Making collaborative system administration easier: Constraints and declarative aspect precedence. In Proceedings of SAICSIT 2004, pages 249–253, Stellenbosch, South Africa, 2004. http://www.fixedpoint.org/lex/papers /holt-2004-mcs.pdf.

[53] G. M. Jones and S. M. Romig. Cloning customized hosts (or customizing cloned hosts). In Proceedings of the 5th Large Installation Systems Administration Conference, pages 233–242, Berkeley, CA, 1991. USENIX.

[54] J. Kephart. Technology challenges of autonomic computing. Technical report, IBM Academy of Technology Study, November 2002.

[55] K. Manheimer, B. A. Warsaw, S. N. Clark, and W. Rowe. The Depot: A framework for sharing software installation across organizational and UNIX platform boundaries. In Proceedings of the 4th Large Installation System Administration Conference, pages 37–46, Berkeley, CA, 1990. USENIX.

[56] J. McDermott. R1: A Rule Based Configurer of Computer Systems. Artificial Intelligence 1982, no. 19, pages 39–80, 1982.

[57] Microsoft. Windows installer technologies: http://www.microsoft.com /windows2000/en/advanced/help/default.asp?url=/windows2000/en/advanced/help /sag_WinInstall_Technology.htm?id=3991.

[58] S. Narain. Configuration management via model finding. In Proceedings of ACM SIGSOFT Workshop on Self-Managed Systems. ACM SIGSOFT, October 2004.

[59] S. Narain, T. Cheng, B. Coan, V. Kaul, K. Parmeswaran, and W. Stephens. Building autonomic systems via configuration. In Proceedings of Autonomic Computing Workshop, June 2004: http://www.argreenhouse.com/papers/narain /Autonomic.pdf.

[60] D. Oppenheimer. The importance of understanding distributed system configuration. Computer Science Division, EECS Department, UC Berkeley: http://roc.cs.berkeley.edu/papers/dsconfig.pdf.

[61] J. Salceda. CIM modelling for system management. Presentation. University of A Coruña, Spain, 2003. http://homepages.inf.ed.ac.uk/group/lssconf/config2003/slides /salceda.pdf.

[62] SysNav. Sysnav white paper. Technical report, SysNav, 2005: http://www.sysnav.com/download.php?id=697354,8,3.

[63] J. Tatemura. Configuration description, deployment, and lifecycle management: XML configuration description language specification. Technical report: http://www.gridforum.org/Meetings/GGF12/Documents/draft-ggf-cddlm-xml-cdl-03.pdf.

[64] E. D. Zwicky. Typecast: Beyond cloned hosts. In LISA VI: 6th Systems Administration Conference, pages 73–78, Berkeley, CA, 1992. USENIX.

About the Author

Paul Anderson (http://www.homepages.inf.ed.ac.uk/dcspaul/) has a background in pure mathematics and over 20 years of experience in system administration. He is currently a principal computing officer with the School of Informatics at Edinburgh University, where he divides his time between research projects in system configuration and the practical problems of managing a complex computing infrastructure. He is the primary author of the LCFG configuration system and the organizer of the LISA configuration workshop series.