THE USENIX SIG FOR

[sage]

SYSADMINS

ISBN-13: 978-1931971522
ISBN-10: 1931971528
90000
9 781931 971522

16

Mark Burgess & Æleen Frisch

A System Engineer's Guide to Cfengine

16 | *Short Topics in*
System Administration

Jane-Ellen Long, Series Editor

A System Engineer's Guide to Host Configuration and Maintenance Using Cfengine

Mark Burgess and
Æleen Frisch

THE USENIX SIG FOR
[sage]
SYSADMINS

16 *Short Topics in*
System Administration

Jane-Ellen Long, Series Editor

A System Engineer's Guide to Host Configuration and Maintenance Using Cfengine

Mark Burgess and Æleen Frisch

Published by the USENIX Association
2007

About SAGE

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

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

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

ISBN 1-931971-52-8

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

The USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA USA 94710
orders@usenix.org
http://www.usenix.org/

First Printing 2007

Contents

5. Extending Cfengine: Modules and Methods 78
5.1 Modules 78
5.2 Methods 79
5.3 Example: Generating the Password and Shadow File 82

Acknowledgments

The authors would like to thank all of the people who have made contributions to this booklet. First, we thank our excellent team of technical reviewers: Chaos Golubitsky, Edward F. Brown, Jason Heiss, and Jeremy Mates. Their very careful reading of the manuscript for this booklet has made it much better. We also thank Kyrre Begnum and Laura Villa for input and pointing out some problems. Finally, we thank all of the members of the cfengine community for their ongoing support and encouragement in the project and in general. Any errors that remain are our own.

1. Introducing Cfengine

As technology becomes more sophisticated,
the cost of introducing variations declines.
—Alvin Toffler, *Future Shock*, 1970

Cfengine is a free software package for automating the configuration and maintenance of networked computers. It is available for all major UNIX and UNIX-like operating systems, and it will also run under recent Windows operating systems via the Cygwin UNIX-compatibility environment/libraries.

Cfengine is suitable for managing anything from one to tens of thousands of hosts. As of this writing, the largest installations we know of regulate around 20,000 machines under a common administration.

Cfengine can manage a great many aspects of system configuration and maintenance, including the following:

- Performing post-installation tasks such as configuring the network interface.
- Editing system configuration files and other files.
- Creating symbolic links.
- Checking and correcting file permissions and ownership.
- Deleting unwanted files.
- Compressing selected files.
- Distributing files within a network.
- Automatically mounting NFS file systems.
- Verifying the presence and integrity of important files and file systems.
- Executing commands and scripts.
- Applying security-related patches and similar system corrections.
- Managing system server processes.

Cfengine's purpose is to implement policy-based configuration management. In practical terms, this means that cfengine greatly simplifies the tasks of system configuration and maintenance. For example, to customize a particular system, it is no longer necessary to write a program that performs each required action in a procedural language like Perl or your favorite shell. Instead, you write a much simpler policy description that documents *how* you want your hosts to be configured. The cfengine software determines what needs to be done in terms of implementation and/or remediation from this specification. Such policy descriptions are also used to ensure that the system remains configured as the system administrator wishes over time.

Here is a brief, annotated example of such a policy description:

Policy Example 1: Introducing Cfengine Configuration

```
control:                                                    General directives.
 tmpdirs = ( tmp:scratch:scratch2 )                       Define a list variable.
 actionsequence = ( files copy tidy )           Specify actions to perform (& order).

files:                               File ownership and protection specifications.
 /usr/local/bin owner=root group=bin mode=755 action=fixall recurse=1

copy:                                                Copy files on/to the local system.
 solaris::                                             Applies only to Solaris systems.
  /config/pam/solaris server=pammaster dest=/etc/pam.d recurse=1
 linux::                                                 Applies only to Linux systems.
  /config/pam/common-auth server=pammaster
  dest=/etc/pam.d/common-auth

tidy:                                          Manage temporary scratch directories.
  /${tmpdirs} include=* age=7 recurse=inf
```

This simple configuration is divided into four stanzas, each introduced by a colon-terminated keyword, specifically**, control:**, **files:**, **copy:**, and **tidy:**. The **control** stanza defines a list of directories, which we've named *tmpdirs* and which we'll use later (in the **tidy** stanza).

The **files** stanza specifies that all of the files in the directory */usr/local/bin* should be owned by user root and group bin and have the file mode 755. When cfengine runs with this configuration description it will correct any ownership and/or permissions that deviate from these specifications. Thus, this stanza serves to implement a policy about the proper ownerships and permissions for the executables in the local binaries directory.

The **copy** stanza prescribes different configurations for Linux and Solaris systems. On Solaris systems, files in */etc/pam.d* will be updated with those in the directory */config/pam/solaris* on a master server when the latter are newer. On Linux systems, only the file */etc/pam.d/common-auth* is updated from the PAM master configuration.[1] Note,

[1] The reason for this is somewhat obscure. Many Linux systems use the PAM **include** file mechanism to propagate this file's PAM stacks to the configuration files for all of the PAM-enabled services. Thus, only this master file will ever change.

however, that both of these specifications implement the same underlying system configuration maintenance policy: update the relevant PAM configuration files from the master server if necessary.

The final, **tidy** stanza illustrates the use of implicit looping. The single directive in the example applies to each of the directories in the *tmpdirs* list. For each directory, cfengine will delete all items in the directory or any of its subdirectories that have not been accessed in seven days (including ones where the filename begins with a period). Like the other directives in this sample configuration file, this stanza implements a policy: items in temporary directories that have not been used within a week will be deleted.

All cfengine configuration descriptions are variations on these and similar themes, albeit more elaborate ones. Before turning to more details about the technical aspects of using cfengine, a brief consideration of the most important underlying and guiding theoretical concepts is in order.

1.1 Fundamental Concepts

As we've stated, cfengine operates on hosts in order to bring their configurations in line with their specified policies. Here are formal definitions of what we mean by these key terms:

> ***Definition 1: Host.*** *Generally, a host is a single computer that runs an operating system like UNIX, Linux, or Windows. We will sometimes talk about machines too, and a host can also be a virtual machine supported by an environment such as VMware or Xen/Linux.*
>
> ***Definition 2: Policy.*** *This is a specification of what we want a host to be like. Rather than being any sort of computer program, a policy is essentially a piece of documentation that describes technical details and characteristics. Cfengine implements policies that are specified via directives of the sort we just considered.*
>
> ***Definition 3: Configuration.*** *The configuration of a host is the actual state of its resources, e.g., the permissions and contents of files, the inventory of software installed, and the like. It is the state of affairs on a particular host at a given time.*

What are we aiming for with cfengine? The answer is, *policy conformant configuration.* We want to formulate a specification for one or more hosts, describing their characteristics and how they all interact (perhaps to solve a business problem); then we want to leave the details, implementation, and maintenance to a robot agent: **cfagent.**

Humans are good at understanding input and thinking up solutions but are not very reliable at implementation: *doing*. Machines and software agents are good at carrying out tasks reliably, but are not good at understanding or finding actual solutions. With cfengine, you let the distinct parts of your human-computer organization concentrate on what they are each good at doing.

Cfengine works at a relatively low level, and it is therefore a pragmatic approach rather than a conceptual approach. Nevertheless, you will find that there are plenty of high-level concepts to think about when deciding on your policy.

1.1.1 Promises, Actions, and Operations

A cfengine policy can be thought of as a list of promises that the system makes to some auditor about its configuration. Most of these promises involve the possibility of *change* to make a host fulfills its policy promises. We call such changes *actions* or *operations.* As you probably already guessed, the auditor in this scenario is part of cfengine itself. Cfagent is also the mechanic or surgeon that performs the operations on the system, if it does not meet its promises.

By describing its operation in this manner, we can think of configuration management as a service, one that is intimately connected with monitoring and maintenance, and which can be "bought" on demand without necessarily subordinating a system to a central authority.

> ***Definition 4: Operation.*** *A unit of change is called an operation. Cfengine deals with changes to a system, and operations are embedded into the basic sentences of a cfengine policy. They tell us how policy constrains a host—in other words, how we will prevent a host from running away.*

For example, here is a promise about the attributes of a file:

```
files:
   /etc/passwd mode=a+r,go-w owner=root group=root action=fixall
```

There are implicit operations (actions) in this declaration: specifically, the operations that will change the attributes if/when they do not conform to this specification.

1.1.2 Convergence

A key property of cfengine is convergence. This is an important characteristic that distinguishes it from general computer languages. It is a property that helps to prevent systems from diverging: running away in an uncontrollable fashion.

> ***Definition 5: Convergence.*** *An operation is convergent if it always brings the configuration of a host closer to its ideal, policy-conformant state and has no effect if the host is already in that state. We can summarize this in functional terms by the following meta-rules:*

$$\text{cfengine(incorrect state)} \rightarrow \text{correct state}$$
$$\text{cfengine(correct state)} \rightarrow \text{correct state}$$

> *We shall sometimes call a "correct state" a "healthy state," using the metaphor that a badly configured host is suffering from a kind of sickness.*

Here is an example used during the editing of an ASCII file:

```
editfiles:
   ...
   AppendIfNoSuchLine "Important configuration line"
```

This operation tells cfengine to append the given text to the end of a file, only if that text is not already there. The policy-conformant configuration is, therefore, that the line is present, and once that is achieved nothing more will be done. We say that the operation **AppendIfNoSuchLine** is convergent.

Don't underestimate the value of convergence. It provides stability. Because cfengine's language interface strongly discourages you from doing anything non-convergent, it also helps to prevent mistakes. The price is that you will have to learn to think in a convergent way—and that is new for most people who come to cfengine for the first time.

1.1.3 Classes and Declarations: From One to Many Hosts

One of the features that make cfengine policies readable is the ability to hide away all of the complex decision-making that needs to be performed by the agent. To realize this ambition, cfengine uses a *declarative* language to express policy.

A declarative language is simply a structured list of sentences (in the case of cfengine, it is a list of policy promises). It is stated in no particular order; it describes a final goal. The details of how one gets there are left implicit, to be evaluated and implemented by the engine that interprets the specification. This is in contrast to *procedural* or *imperative* languages, such as shell or Perl, which micro-manage every step along the way.

In an imperative language, one focuses on the procedure. In a declarative language, one focuses on the intention or the presumed result.

One example of this is the use of *classes* in cfengine. Classes are a way of making decisions without writing many "if-then-else" clauses. A class is an identifier, a Boolean variable that has the value "true" when a particular test is true. In other words, it caches the result of an "if" test. A class is used to limit the scope of cfengine actions, to the appropriate system(s) and/or under the appropriate conditions.

The benefit of classes is that all of the testing can be hidden away in the bowels of cfengine, and only the results need be visible if or when they are needed.

> ***Definition 6: Classes.*** *A class is a way of slicing up and mapping out the complex environment of one or more hosts into regions that can then be referred to by a symbol or name. They describe scope: where something is to be constrained.*

For example, the class **debian** is true if and only if cfagent is running on a host that has Debian GNU/Linux as its operating system.

1.1.4 Voluntary Cooperation

It is a fundamental property of cfengine components that every host retains its individual autonomy. A host can always opt out of cfengine-based governance if its administrator wants it to. This principle leads to a fundamental design and implementation decision:

> ***Definition 7: Autonomy.*** *No cfengine component is capable of receiving information that it has not explicitly asked for itself.*

It is important to understand what this means. It does not mean that centralized control of hosts cannot be achieved. Centralized control is the way most administrators choose to use cfengine. Indeed, all you have to do to achieve centralized control is to make a policy decision for all your hosts to fetch policy specifications from a central authority.

Autonomy does mean that if your environment has some small groups or sub-cultures with special needs, it is possible for them to retain their special identity. No self-appointed authority can ride roughshod over their local decisions.

Where does policy come from, then? Each host works from a policy specification that cfengine expects to find in a local directory (usually */var/cfengine/inputs* on a UNIX-like host). If you want your host to be controlled from some central manager or authority, then your policy must contain bootstrapping specifications that say, “It is my decision that I should download and follow the policy specification located at the central manager.”

Each host can turn this policy decision off at any time. This is a key part of the cfengine security model.

1.1.5 Scalability

Cfengine is designed to be maximally scalable. Its scalability is at least as good as any other system, because it allows for maximal distribution of workload.

> ***Definition 8: Scalable distributed action.*** *Each host is responsible for carrying out checks and maintenance on/for itself, based on its local copy of policy.*

This does not mean that you are immune from making bad decisions. For example, network services can always be a bottleneck if you ask 10,000 hosts to fetch something from one place at the same time.

The fact that each cfengine agent keeps a local copy of policy (regardless of whether it was written locally or inherited from a central authority) means that cfengine will continue to function even if network communications are down.

1.2 Cfengine Components

The cfengine software consists of a number of components, separate programs that work together (see Figure 1.1).[2]

The components of cfengine are:

- **cfagent:** Interprets policy promises and implements them in a convergent manner. The agent can use data generated by the statistical monitoring engine **cfenvd**, and it can fetch data from **cfservd** running on local or remote hosts.
- **cfexecd**: Executes **cfagent** and logs its output (optionally sending a summary via email). It can be run in daemon (standalone) mode, or it can be run from **cron** on a UNIX-like system.
- **cfservd**: Monitors the cfengine port: serves file data and starts **cfagent** on receipt of a connection from **cfrun**. Note that no data can be passed to this daemon.
- **cfrun**: Contacts remote hosts and requests that they run **cfagent**.
- **cfenvd**: Collects statistics about resource usage on each host for anomaly detection purposes. The information is made available to the agent in the form of cfengine classes so that the agent can check for and respond to anomalies dynamically.
- **cfkey**: Generates public-private key pairs on a host. You normally run this program only once, as part of the cfengine software installation process.
- **cfshow**: Dumps the **cfagent** database contents in ASCII format, in case you ever become interested in its internal memory.
- **cfenvgraph**: Dumps **cfenvd**'s statistical database contents in a form that can be used to plot graphs showing the normal behavior of a host in its environment.

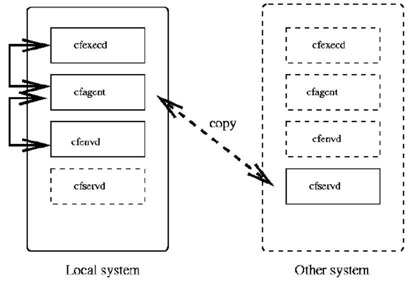

Figure 1.1: Cfengine Components and the Connections Between Them

[2] The components differ between version 1 and version 2. We shall only discuss cfengine 2 here, as cfengine version 1 is no longer supported, and you are strongly advised to use version 2. In addition, cfengine version 3 is being developed at the time of writing, but this will take a number of years before it can fully replace version 2. It will incorporate the state of the art in network and system sdministration research, building on all the lessons learned from versions 1 and 2.

Figure 1.1 illustrates the relationships among cfengine components on different hosts. On a given system, **cfagent** may be started by the **cfexecd** daemon; the latter also handles logging during **cfagent** runs. In addition, operations such as file copying between hosts are initiated by **cfagent** on the local system, and they rely on the **cfservd** daemon on the remote system to obtain remote data.

1.3 Getting Started

In this section, we'll get cfengine installed and running. You should get the cfengine components working with a trivial policy before trying to understand the details of the language, just to get the engine ticking over. Later, when you have understood its operation, you can build up your policy step by step.

1.3.1 Building the Software

Cfengine is installed like most other UNIX open source software. You can either use a packaged version that comes with your operating system, or you can compile it from source code.

In either case, you will need two libraries: BerkeleyDB, for internal database usage, and OpenSSL for cryptographic methods. These libraries are both open source and can be used freely, just like cfengine. You cannot use cfengine without these libraries, nor can you replace the libraries with something else.[3]

To begin the cfengine installation process, download the source code archive from *http://www.cfengine.org* or one of its mirror sites. The download will be a compressed tar archive with a name of the form *cfengine-2.x.x.tar.gz*, where *x.x* indicates the minor version number within cfengine version 2.

The following procedure summarizes the steps required to build cfengine:

Procedure 1: Installing Cfengine from Source Code

```
$ tar zxf cfengine-2.x.x.tar.gz
$ cd ./cfengine-2.x.x
$ ./configure
$ make
$ sudo make install
```

The default location for installed binaries on a UNIX-like host is under */usr/local/sbin*.

[3] The databases that cfengine uses are fast low-level structures for internal memory. They are not used for user data storage. Rather, cfengine needs a local database with very fast lookup capabilities. For this reason, SQL relational databases are unsuitable and cannot be used with cfengine.

This directory is sometimes a shared file system (e.g., on a remote file system mounted via NFS). This could be a problem, as cfengine must be able to function even if the network is down. For this reason, copies of the cfengine binaries are maintained within the cfengine directory tree.

1.3.2 Setting Up Your First Cfengine Host

Eventually, you will be able to let cfengine do most of the work of installing itself on new computer systems. As a novice, however, you will want to learn something about how things work. To accomplish this, we will now install cfengine manually so that everything is manifest.

To avoid unnecessary dependencies on network file systems, cfengine uses a directory that is guaranteed to be local on any host (except diskless clients). The default location (referred to as the "work directory") is */var/cfengine*. At this stage, we'll also assume that the cfengine binaries are installed in */usr/local/sbin*.

The next step is to create the basic structure of the cfengine work directory tree:

Procedure 2: Creating Cfengine Work Directories Manually

```
# mkdir /var/cfengine
# mkdir /var/cfengine/bin
# mkdir /var/cfengine/inputs
```

Next, make local copies of the cfengine binaries in the work directory's *bin* subdirectory (i.e., */var/cfengine/bin*). These are the copies that are actually executed, so that there will be no risk of having a network hang during execution.

Procedure 3: Copying Cfengine Binaries to the Work Directory

```
# cp /usr/local/sbin/cfagent /var/cfengine/bin
# cp /usr/local/sbin/cfexecd /var/cfengine/bin
# cp /usr/local/sbin/cfservd /var/cfengine/bin
# chown -R root:0 /var/cfengine
# chmod -R 755 /var/cfengine
```

Now that the binaries are in a reliable location, let's test the agent by creating a trivial cfengine policy.

Create the following file as */var/cfengine/inputs/cfagent.conf*:

Policy Example 2: Trivial Policy for Initial Testing

```
#/var/cfengine/inputs/cfagent.conf
control:
   actionsequence = ( shellcommands )

shellcommands:
   "/bin/echo Danger, Will Robinson!"
```

This is all you need to test cfengine. The policy is a simple one: it simply promises to print out a message. Test this now by running the agent. The agent will look for the *cfagent.conf* file in the work directory by default. Note also that we need to run the **cfkey** command once,[4] prior to the first time that we run **cfagent**.

Procedure 4: Run the Agent to Test Cfengine's Basic Functioning

```
# /usr/local/sbin/cfkey                    Run once, before your first cfagent command.
# /var/cfengine/bin/cfagent
cfengine::/bin/echo Dange: Danger, Will Robinson!
```

Now for a surprise! Run the **cfagent** command above a second time, immediately afterwards, and you will see that nothing happens. This is normal. In fact, nothing more will happen until at least a minute has elapsed from the last time you ran cfengine. If you run **cfagent -v**, invoking verbose mode, you will see the message within the output:

```
cfengine:: Nothing scheduled for
[shellcommand./bin/echo Danger, W] (0/1 minutes elapsed)
```

This message is telling you that cfengine thinks it is too soon to repeat this promised action. We'll return to this matter later, when discussing cfengine's transaction locks.

Congratulations: you have now successfully used cfengine.

1.3.3 Creating a Permanent Setup

It is normal to have cfengine run on a regular basis: once per hour or perhaps every 15 minutes, depending on your requirements. Running **cfagent** often need not be a burden on the system, because the agent will not repeat actions unnecessarily (keep in mind the property of convergence).

We could accomplish this by editing a *crontab* file manually, but, instead, let's use cfengine to do it for us.

[4] This command creates the public/private key pair for the local system, storing the resulting files in the */var/cfengine/ppkeys* subdirectory. It also creates the *randseed* file and several additional subdirectories in the cfengine work directory.

Edit the policy file so that it matches the following example:

Policy Example 3: Cron Runs cfexecd Every 15 Minutes

```
# /var/cfengine/inputs/cfagent.conf
control:
   actionsequence = ( editfiles )
   EmailTo = ( sysadmin@mydomain.tld )

editfiles:
  !SuSE::
   { /var/spool/cron/crontabs/root
     AutoCreate
     AppendIfNoSuchLine
       "0,15,30,45 * * * */var/cfengine/bin/cfexecd -F"
   }

  SuSE::
   { /var/spool/cron/tabs/root
     AutoCreate
     AppendIfNoSuchLine
       "0,15,30,45 * * * * /var/cfengine/bin/cfexecd -F"
   }
```

We have removed the Lost in Space policy, replacing it with one about root's *crontab* file. Instead of (possibly) running a shell command, the policy can potentially perform some simple file editing. In addition, it makes some references to SuSE Linux in the lines ending with double colons. These expressions are another example of cfengine classes, and they determine when the promises that follow apply.

In the example, we are taking account of the fact the SuSE Linux uses a different directory convention for *crontab* files than most other operating systems. So we make a rule for SuSE hosts and another rule for non-SuSE hosts (as you might guess, "!" means logical NOT), with the latter appearing first in our sample policy file.

In both cases, the promise is to add a line to *crontab* if it does not already exist.[5] We'll explain file editing in more detail later in this booklet.

The revised policy file also includes an additional directive in the **control** section. It specifies the target address for email generated by **cfagent** when initiated by **cfexecd**. In

[5] This simple example is not as careful as a real one would need to be in at least two respects. First, not all non-SuSE UNIX and Linux systems use the specified location for *crontab* files, so the second class expression needs to be more complex for many environments. Second, the editing directives should take more care to ensure that multiple entries for **cfexecd** do not appear in the final *crontab* file.

general, output from such **cfagent** runs are stored in timestamped files in */var/cfengine/outputs*, and this subdirectory is created by **cfexecd** if it does not already exist.

1.3.4 What's Next?

Starting from this simple policy being enforced on a single host, you can build up your cfengine implementation, expanding it both to include more hosts and to place more aspects of system configuration and maintenance under cfengine control. We will consider these two activities separately in the chapters that follow.

2. Cfengine Policies: Under the Hood

In this chapter, we'll consider creating policies for cfengine in detail. As a user of cfengine, you naturally want to move beyond simple recipes to a state of understanding, since understanding and mastering tools are prerequisites for ensuring the predictability—and, therefore, security—of your site.

Cfengine uses a language interface for maximally expressive policy specifications. The language has a very simple, free-format grammar which has grown organically as the cfengine research project has progressed. This has made it somewhat inconsistent at times, and the goal of simplicity has brought with it a few limitations in the parser.[6]

At the highest level, the cfengine policy grammar has the following simple form:
rule-type **:**

[*class-expression*::] *Classes are optional, defaulting to the "any" class.*

policy rule 1
policy rule 2
...
policy rule n

Not all rules can be used in all contexts. The rule types used in the *cfservd.conf* file are rather different from the ones that are valid in *cfagent.conf*, and the file *cfrun.hosts* has an entirely different format (we'll cover each of these files in due course).

Most rules follow this general structure:

target option=value option=value ...

where the various options control when and how the target item is validated and/or modified.

The following are the most important characteristics of the cfengine policy language:

- ❖ Free-format language, with some parser restrictions. Rules can extend over as many lines as necessary or desired. Indentation is conventionally used for readability.
- ❖ Internal setting and macro (local variable) values are enclosed in parentheses. Built-in setting values may consist of multiple items, separated by spaces.

[6] These limitations will not be removed in version 2 of cfengine; they will be dealt with comprehensively in version 3, via substantial modifications to the existing syntax.

- Locally defined lists are enclosed in parentheses as well, with list elements separated by colons.[7] However, any desired separation character may be defined for this context using the **Split** directive in the **control** section.
- Variables and lists are dereferenced using the following syntax: *${name}*.[8]
- Comments use the shell syntax; a # sign marks the remainder of the line as a comment.

Here is a brief excerpt from a *cfagent.conf* file that illustrates several of these features. It contains some initial settings and definitions in its **control** section, and two policy rules: one of type **files** and one of type **tidy**.

```
control:
   domain = ( cfengine.org )
   ChecksumDatabase = ( /var/cfengine/db/cfdb )
   actionsequence = ( tidy files )
   Split = ( , )
   tmpdirs = ( tmp,scratch,aux/temp )
   maxage = ( 3 )

# File ownership and Tripwire-like checksum verification
files:
  Hr02.linux::  # Linux systems during the 2AM run
   /sbin owner=0 group=0 mode=o-w checksum=md5 recurse=inf

tidy:
   /${tmpdirs} include=* age=${maxage} recurse=inf
```

The **control** section defines four cfengine settings: the local domain, the location of the database of checksum values referred to in the **files** rule, the ordering for the two policy stanzas that follow, and the item separation character for lists (here, a comma).[9] The section's final two lines define a list named *tmpdirs* and a macro named *maxage*, which are used in the **tidy** stanza.

The **files** section illustrates the use of comments and of a compound class expression: in this case, two classes joined by logical AND, denoted by a period. The rule in this section checks the user and group owners and owner write permission for all of the files under */sbin*. It also computes an MD5 checksum for each file and compares it to a stored value in */var/cfengine/db/cfdb* (the database location is specified in the **control** section). The agent will report on any file with incorrect ownership, user write permission, or an incorrect checksum.

[7] In **cfagent** policy files. The default separator in *cfservd.conf* is the comma.

[8] Parentheses are legal replacements for the curly braces, but a parser bug in cfengine version 2 makes them unreliable. We therefore recommend avoiding them.

[9] Careful readers will notice that the **actionsequence** items do not use any defined split character but, rather, are always separated by spaces.

The **tidy** section is similar to the one we considered in the previous chapter. In this case, the rule removes all files that have not been accessed in three days from the */tmp*, */scratch*, and */aux/temp* directory trees. The rule uses the *maxage* macro for the value of the access time limit option, **age**.

We'll now take a look at the most important rule types and then return to some more advanced features of the configuration language.

A note about configuration snippets: Not all of our illustrative examples are runable as is by cfengine. For example, many of them will not include **actionsequence** directives or even **control** stanzas at all. You may need to modify them in order to incorporate them within your own configuration files. All examples assume cfengine version 2.2.0 or later.

2.1 Action Sequence Rule Types

We've already introduced you to several of the rule types supported by **cfagent** configuration files (e.g., *cfagent.conf*) . In this section, we will discuss the most widely used of these in some detail.[10] Table 2.1 briefly describes the rule types that appear in the following subsections.

actionsequence Item	Purpose
alerts	Display messages (based on classes defined).
copy	Copy files to or update files on the local system; the source files can be local or remote.
disks	Verify presence of/free space on disk partitions.
disable	Deactivate system features by renaming configuration files. This rule type can also perform log file rotation.
editfiles	Modify text files (typically, system configuration files).
files	Verify/correct the attributes of files. This rule type can also specify file compression.
links	Verify/create/correct the attributes of symbolic links.
netconfig	Configure the network interface in the **defaultroute** and **interfaces** stanzas.
resolve	Specify name servers and other *resolv.conf* contents.
packages	Verify the presence of/install software packages.
processes	Monitor and manage processes.
shellcommands	Execute external shell commands.
tidy	Deleted unwanted files and directories.

Table 2.1: Principal Cfengine Rule Types

[10] The discussion of **alerts** is postponed until we discuss classes, later in this chapter.

Unfortunately, space limitations do not allow us to cover all of the available rule types here. Consult the *Cfengine Reference* (*www.cfengine.org/docs/cfengine-Reference.html*) for complete information about all rules types and options.

2.1.1 Important Options

We'll begin by discussing some options that are available for many different rule types. We'll see examples of these options in action as we consider each rule type in turn. Figure 2.1 illustrates the availability of these options by rule type.

	GENERAL OPTIONS SUPPORTED							
Rule Type	**inform/ syslog**	**recurse/ xdev**	**define/ elsedefine**	**ifelapsed/ expireafter**	**include**	**exclude**	**ignore**	**filter**
copy	√	√	√		√	√	√	√
disks	√			√				
disable	√		√					
editfiles	√‡	√†	√‡		√	√	√	√
files	√	√	√	√	√	√	√	√
links	√	√	√	√	√	√	√	√
packages			√	√				
processes	√		√	√	√	√		
shellcommands	√		√	√				
tidy	√	√	√	√	√	√	√	√

†No **xdev** option. ‡Feature is present using an alternate syntax.

Figure 2.1: Availability of Commonly Used Options by Rule Type

Logging Options

There are two options that control whether and where **cfagent** reports on its activities. By default, most actions are performed silently.

- **inform=on**: Report on all actions taken (to standard output).
- **syslog=on**: Log all actions taken to **syslog**. You can specify the desired facility with the **SyslogFacility** setting in the **control** section (the default is LOG_USER).

There are also **control** sections, which can be used to enable and disable **inform** and **syslog** everywhere they are supported.

Here are some sample directives illustrating these features:

```
control:
   Inform = ( off )
   Syslog = ( on )
   SyslogFacility = ( LOG_LOCAL1 )

disable:   # Rename files => deactivate features
   /etc/hosts.equiv inform=on   # Report and log to syslog
   /etc/ftpusers syslog=off    # Disable silently
```

Specifying Successive Run Timeouts

The following options specify timeout periods for **cfagent** actions:

- **ifelapsed=***m*: Ignore rule unless at least *m* minutes have passed since the previous run. The default is 1 minute.
- **expireafter=***n*: Assume that the action corresponding to the rule is hung after *n* minutes (i.e., its maximum lifetime). It will be killed by any later **cfagent** run.

There are also **control** settings which correspond to global values (i.e., defaults) for these options. Here are some example of these settings and options:

```
control:
   IfElapsed = ( 10 )
   IExpireAfter = ( 15 )

copy:  # Copy RPM files at most once an hour.
   I/rpm_out dest=/rpm_in server=silo include=*.rpm
     ifelapsed=60 recurse=inf
```

This rule will result in server *silo* being checked for RPM files at most once an hour (the actual frequency depends on how often **cfagent** and **cfexecd** are executed). Any copy operation that results will be assumed to be hung after 15 minutes. Other rules in the same configuration file as these lines could potentially be deployed every 10 minutes.

Controlling Directory Tree Traversal

There are two options which control how subdirectories encountered during an operation are handled. By default, rules apply only to the specified items; in other words, actions are not recursive by default.

- **recurse=***depth*: Perform recursive checks/operations, descending at most *depth* levels. Use a depth value of 1 to act on the items within a specified directory. Use the keyword **inf** to descend to the bottom of the directory tree.
- **xdev=off**: Descend into subdirectories residing on different disk partitions. By default, partition boundaries are not crossed.

Here are some examples of these options:

```
files:  # Check ownerships under /usr/local
   /usr/local owner=root group=admin mode=755 recurse=inf
```

```
tidy:  # Clear /tmp and subdirectories (>3 days old)
   /tmp age=3 include=* rmdirs=sub exclude=.X11
    recurse=inf xdev=off
```

The first rule checks the user and group owners of files in the */usr/local* directory tree, reporting on any that are incorrectly set. The second rule removes files and empty subdirectories that have not been accessed in three days under */tmp* (except the *.X11* subdirectory), regardless of the disk partition on which the items reside.

See the discussion of the **ignore** option in the next subsection for another method of controlling directory tree traversal.

File and Directory Inclusion and Exclusion Patterns

- **include**: Include items matching the specified patterns when selecting files and/or directories for verification or modification. Patterns may include the shell wildcards ***** (match any characters, including no characters) and **?** (match any one character).
- **exclude**: Exclude items matching the specified patterns when selecting files and/or directories for verification or modification. Global exclusion lists can be specified for copying and linking operations via the **ExcludeCopy** and **ExcludeLink** settings, respectively, in the **control** section.
- **ignore**: Ignore items matching the specified patterns. In contrast to **exclude**, directories matching an item in the **ignore** list are not traversed during recursive operations. A global list of directories to ignore can be specified via the separate **ignore** stanza (see the example below).
- **filter**: Select items to which to apply a rule based on complex filtering criteria. Filters are discussed in detail later in this chapter.

Here are some brief examples of some of these settings and options:

```
control:
   ExcludeCopy = ( *.bak *~ )

ignore:
   .XII
   /usr/local

tidy:  # Remove non-recent files from /tmp and /scratch
   /tmp age=1 include= * recurse=inf
   /scratch age=1 include=* exclude=*.sav recurse=inf
```

```
copy:  # Update local documentation from server silo
   /masterdoc dest=/usr/local/doc server=silo recurse=1
files:
   /usr recurse=inf action=fixall owner=root
```

This example specifies a global exclusion list for copy operations and a list of subdirectories to ignore during recursive operations. The **tidy** rule will clean up files that haven't been accessed today from */tmp* and all of its subdirectories except */tmp/.X11* (the location of X11 semaphores). It will also remove such files from the */scratch* directory, except ones having the extension **.sav**.

The copy rule copies all files from *silo:/masterdoc* that are newer than the version in */usr/local/doc* (if any), excluding any whose name ends in a tilde character (emacs backup files) or that have the extension *.bak* (using the global copy exclusion list.) Note that having */usr/local* in the directory **ignore** list does not affect the file copying operation, since the former applies only to directory traversal in recursive operations. In contrast, it is excluded from the **files** operation.

Additional Options

The remaining options appearing in Figure 2.1—**define**, **elsedefine**, and **filter**—will be discussed later in this chapter.

2.1.2 Configuring the Network Interface

Cfengine provides several rule types that enable the network interface to be minimally configured. They are illustrated in the following policy definition:

Policy Example 4: Network Interface Configuration

```
control:
   domain = ( cfengine.org )
   actionsequence = ( netconfig resolve )

interfaces:                      Corresponds to the netconfig actionsequence item.
  linux::
   "eth0" netmask=255.255.255.0 broadcast=ones

defaultroute:
   192.168.1.1

resolve:
   192.168.1.100
   192.168.1.105
```

These rules specify the netmask and broadcast address used by the specified Linux network interface, as well as the default route (router address). They also configure the */etc/resolv.conf* file, specifying the local domain and DNS name servers. When **cfagent** runs, the IP addresses specified in the **resolve** stanza will be listed as the first two name server entries in *resolv.conf*, and the domain defined in the file will be the one specified in the **control**. Thus, *resolv.conf* on this system will look like this:

```
domain cfengine.org
nameserver 192.168.1.100
nameserver 192.168.1.105
```

Any existing nameserver entries

If you would like the *resolv.conf* name server entries to consist only of the listed items, then include the following setting in the **control** section:

```
control:
   EmptyResolvConf = ( true )
```

Note that the network interface rules we've just considered are convergent. For example, if */etc/resolv.conf* already has the specified configuration, then no modifications will be made to it. Similarly, once the network interface is properly configured, subsequent **cfagent** runs will simply verify that its configuration is correct. Remediation will be necessary only if something untoward happens to the network interface in the interim.

2.1.3 Monitoring and Protecting Files and Directories

In this subsection, we'll consider rule types which enable you to monitor the contents and attributes of important system files and directories and to take certain kinds of corrective actions. The **files** rule type allows you to check and correct directory and file permissions, and the **links** rule type maintains symbolic and hard links.[11]

The following example rules illustrate some ways that **files** can be used to maintain the file system in a correct configuration:

Policy Example 5: Checking File/Directory Ownerships and Permissions

```
files:
   /usr/local/sbin owner=0 group=admin mode=755 recurse=1
   /usr/local/bin mode=ugo-w owner=root,bin,admin
     action=fixall recurse=1
   /home mode=g-w,-6000 action=fixplain inform=true
     recurse=inf
```

[11] Actually, both of them do more than this, but this description is a good place to start.

The first rule will report on any items within */usr/local/sbin* whose ownership or permissions differ from the ones specified.

In the second rule, the same attributes are checked for the files in */usr/local/bin*, and the **action** option tells cfengine to correct any deviating settings. The rule specifies that write permission should not be set for any item in any context; the syntax of the **mode** option is quite flexible and generally follows that used by **chmod**. The **owner** option has a list of usernames specified as its argument. The user owner for any file whose current user owner is not included in this list will be set to the list's first item, "root." Such a list is also valid for the **group** option.

The third rule illustrates some additional features of the **mode** option. The hyphen preceding the octal mode value means that the corresponding bits should be turned off (and a plus sign means "turn on"), and more than one item may be included in the option's argument. Thus, this rule has the effect of removing **setuid** and **setgid** permissions for all plain files under */home* as well as group write access, reporting on each action that results.

As we've seen, by comparing checksums cfengine can also determine whether file contents have been modified. The following policy illustrates the method for initializing or updating the checksum database used by this feature:

Policy Example 6: Initializing/Updating the Checksum Database

```
control:
#  Compute/store checksums; remove entries for deleted files
   ChecksumDatabase = ( /var/cfengine/db/cfdb )
   ChecksumUpdates = ( on )
   ChecksumPurge = ( on )
   bindirs = ( bin:usr/bin:sbin:usr/sbin:usr/local/bin )

files:
   /${bindirs} checksum=md5 recurse=inf
```

Once the database values have been initialized, best practice is to copy the database to read-only media (such as CD-ROM), which can be mounted in the same location.

The following rule will compare the current checksums for the files under these directories to the ones stored via Policy Example 6, reporting on any discrepancies:

```
control:
   ChecksumDatabase = ( /var/cfengine/db/cfdb )
   bindirs = ( bin:usr/bin:sbin:usr/sbin:usr/local/bin )

files:
   /${bindirs} checksum=md5 recurse=inf action=warnplain
```

The **files** rule type can also monitor and modify certain operating- system specific attributes, including Solaris and Windows access control lists (ACLs) and BSD security flags, as in these examples:

Policy Example 7: Specifying Solaris ACLs and BSD Flags

```
acl:      # Define Solaris ACL
    {  secure
       method:overwrite
       fstype:posix
       default_user:*:=rwx
       default_group:chem:=rwx
       default_other:*:=
       user:chavez:=rwx
       user:mark:+rx
       user:kyrre:=r
       mask:*:rwx
    }

   files:
     solaris::  # Apply ACL to files
       /private acl=secure action=fixall recurse=inf

     bsd::      # Ensure that the immutable flag is set
       /special flags=uchg action=fixall recurse=inf
```

Finally, **files** rules can also specify file compression, file and directory creation, and modification time updating, as in these examples:

```
control:
   CompressCommand = ( /usr/bin/gzip )

files:
   /depot include=*.tar action=compress recurse=inf
   /var/log/messages owner=root group=0 mode=755 action=create
   /scratch/ owner=root group=root mode=777 action=create
   /usr/local/src/rabbit include=*.c action=touch recurse=1
```

The first rule compresses all files in */depot* with the extension *.tar* (presumably **tar** archives). The second rule creates the file */var/log/messages* if it does not already exist, with the specified ownerships and mode; if the file does exist, its attributes are unmodified. The third rule similarly creates the */scratch* directory if necessary (note the final slash in the rule "*/scratch/*").[12] The final rule updates the modification times of all C source files in the specified directory.

The **links** rule type is primarily used for maintaining required links.[13] By default, such rules specify symbolic links whose targets are absolute pathnames, contrary to the workings of the UNIX **ln** command. The following policies illustrate this rule type:

Policy Example 8: Maintaining Links

```
links:  # Specify required links
   /logs -> /var/log
   /etc/aliases -> /etc/postfix/aliases type=relative
   /mascot.jpg -> /usr/local/lib/images/ahania.jpg type=hard
   /home/g03 ->! /homes/mike/g03  # Force link to conform
```

The first, second, and fourth rules specify symbolic links that should be present, while the third rule specifies a hard link. The second rule specifies a symbolic link expressed as a relative pathname, i.e., *./postfix/aliases.*

When processing **links** rules, cfengine checks the links and their targets, adding missing links that have valid targets and reporting on links that have the wrong target, point to a nonexistent item, or are themselves plain files or directories (rather than links). In addition, you can force link rules to be enforced in all circumstances by including an exclamation point in the specification, as in the final example above. Table 2.2 charts how link rules are applied, normally and when forced.

Link Exists?	Target Status	Default Action	Forced Action
No	Exists	Link created	Link created
No	Missing	Warning	Link created
Yes	Correct	None	None
Yes	Wrong item	Warning	Link corrected
Yes	Missing	Link removed	None
Wrong type	Exists	File renamed to *name.cfsaved*; link created	File renamed to *name.cfsaved*; link created
Wrong type	Missing	Warning	File renamed; link made

Table 2.2: Link Rule Effects (Default and Forced)

[12] The **directories** rule type can also be used for this function.

[13] It can also be used to perform file copying in some circumstances, but we will not cover that capability here.

2.1.4 Managing Configuration Files (and Others)

Cfengine provides two types of rules that are very useful for managing configuration and other files on a system. The **disable** rule type renames files, typically by adding the extension *.cfdisabled*; such renaming can often serve to disable unwanted services and features. The **editfiles** rule type enables you to modify the contents of system configuration files and other ASCII text files.

The following policy illustrates these rule types:

Policy Example 9: Disabling and Modifying Files

```
disable:
  # Disable passwordless system access
  /etc/hosts.equiv
  /root/.rhosts inform=true
  # Rotate syslog file; truncate maillog when big
  /var/log/messages rotate=6 ifelapsed=1440
  /var/log/maillog rotate=truncate size=>1024m

editfiles:
  { /etc/hosts.allow # Disable access for this domain
    HashCommentLinesContaining "bad-guys.org"
  }

  { /etc/xinetd.d
    Backup "false"
    # Make sure telnet is disabled
    BeginGroupIfLineMatching ".*telnet.*"
     DeleteLinesMatching " *disable =.*"
     GotoLastLine
     InsertLine "disable = yes"
    EndGroup

    # If no access control, limit to subnet
    SetLine "only_from = 192.168.9"
    AppendIfNoLineMatching " *only_from.*"
  }
```

There are four rules in the **disable** stanza. The first two rename the specified files by adding the aforementioned extension, thereby deactivating certain types of passwordless remote system access.

The **disable** type can also be used to rotate system log files (like the **logrotate** facility). The third rule in the **disable** stanza maintains the most recent six versions of the *mes-*

sages log file, renaming older ones to *messages.1* through *messages.5*. This file will be rotated at most once per day (controlled by the **ifelapsed** option).[14] The final rule in the stanza truncates the *maillog* file whenever its size exceeds 1 GB.

The **editfiles** rule type allows you to modify the contents of text files, using a rich set of editing primitives. The first rule in Policy Example 9 will place a comment character—in this case, a pound sign (#)—at the beginning of any active line in */etc/hosts.allow* in which the string "bad-guys.org" appears.

The second rule in the **editfiles** stanza loops over all of the files in the */etc/xinetd.d* directory. The first directive disables the creation of backup files containing the previous version of modified files. The next group of editing commands apply only to a file having a line that matches the specified regular expression—i.e., containing the string "telnet" somewhere within it—as indicated in the **BeginGroupIfLineMatching** directive; only the file for the **telnet** service should contain a matching line. For this file, the editing operations will result in the **disable** keyword appearing as the last entry within the file, with a setting of "yes."

The remaining editing operations in this section will be applied to every file within the specified directory. The **SetLine** operation defines the specified character string as the active replacement text. The second operation causes this text to be placed at the end of the file when the string "only_from" does not appear anywhere within the file. These operations have the effect of adding the specified **only_from** setting to any file that does not already have one.

These **editfiles** examples illustrate just a few of the many available editing operations. Consult the *Cfengine Reference* for information about the entire set.

Note that **editfiles** does not modify the file if the specified directives do not result in any actual changes to the file, and the file's timestamps do not change. This is true even if directives such as **EmptyEntireFilePlease** are used (discussed below).

When you are constructing **editfiles** rules, it is important to be certain that the operation sequence you are specifying is convergent and will always result in the same final form of the file. It is all too easy to introduce unintended assumptions about the specific form of the configuration file which may be violated in practice. Sometimes the best way to ensure that a file has the desired contents is to replace it completely, either by copying a saved version or with an **editfiles** policy such as the following:

```
editfiles:
   { /etc/issue  # Specify entire file contents.
      EmptyEntireFilePlease
      InsertLine "Welcome to my parlor! Please login."
   }
```

[14] We'll see other ways of accomplishing this when we consider classes, later in this chapter.

Note that **editfiles** does not create files which do not already exist unless the **AutoCreate** directive is included.

Using a Central Repository

We have seen several contexts where cfengine will save the current version of a file that it is modifying or replacing. By default, such files are given a new extension and remain within the directory in which they were encountered. Alternatively, you can specify a repository directory to which such files can be moved instead. The repository location is specified in the **control** section:

```
control:
   Repository = ( /var/spool/cfengine )
```

Files moved to the repository are given names reflecting their full paths, with slashes replaced by underscore characters.

The repository is used by **disable**, **editfiles**, **links**, and **copy** rule types; **copy** and **disable** allow you to override repository use or to specify an alternate repository directory via their **repository** option.

2.1.5 Copying and Distributing Files

We've seen several examples of cfengine's ability to copy files already. In fact, file distribution within a network is one of cfengine's most powerful and widely deployed capabilities. Rules related to these activities appear in the **copy** stanza.

By default, file copy operations are performed subject to the following conditions:

- A file is copied only when the source file is newer than an existing destination file. This is determined by comparing their **ctime**s,[15] but alternative criteria can be specified using the **DefaultCopyType** global setting or with the **type** option (for individual rules). Some useful comparison types are **mtime** (content modification times), **checksum** (computed checksums), and **binary** (binary file comparison). You can also force a copy operation to take place even when the comparison says it doesn't need to, by including the **force=on** option.
- Old versions of copied items are retained, either by giving them the extension *.cfsaved* or moving them to the repository. You can force existing items to be simply replaced, by including the **backup=off** option.
- The modification and access times for copied files reflect the time when the copy operation took place. You can force copied items to retain the same timestamps as the source files by including the **timestamps=on** option.

[15] The UNIX "ctime" is the inode change time, and it is updated whenever the corresponding file's contents or attributes are changed.

- Cfengine will refuse to perform copy operations that modify the item type of the target (for example, replace a directory with a plain file). This behavior can be overridden with the **typecheck=off** option.
- When copying complete directories, files present in the target location but not in the source directory are ignored. Such files can be automatically deleted using the **purge=on** option, in order to maintain identical directories.
- Copy operations do not verify the resulting data. Verification is performed (via a checksum comparison) when the **verify=on** option is specified.
- Network copy operations transmit data in the clear. For sensitive information, you can include the **encrypt=on** option, which causes network transmissions to be encrypted.

The following sample policies illustrate some of the **copy** rule type's capabilities, including some of the options we just considered.

Policy Example 10: File Copying Policies

```
control:
   DefaultCopyType = ( mtime )
   SplayTime = ( 15 )
   adminhost = ( secrets.cfengine.org )

copy:
#  Copy dat/doc files if not too big
   /usr/local/data dest=/archive/data
     include=*.dat include=*.doc exclude=test.*
     recurse=inf backup=false size=<500m

#    Retrieve configuration file from master
   /depot/hosts.deny server=${adminhost}
     dest=/etc/hosts.deny owner=root group=0 mode=644
     backup=off force=on timestamps=keep

#    Transmit shadow password file encrypted
   /depot/shadow server=${adminhost} dest=/etc/shadow
     owner=0 group=0 mode=600 encrypt=true
```

The first rule specifies that *.dat* and *.doc* files within the */usr/local/data* directory tree be copied to */archive/data*, provided that the source files have been modified more recently then their counterpart in the target directory and that they are smaller than 500 MB. In addition, files having the name *test* are also excluded. Existing files will be overwritten without being saved.

The second rule unconditionally replaces the local */etc/hosts.deny* file with one from the system *secrets.cfengine.org*, retaining the timestamps from the source file. This rule also specifies the ownership and mode for the target file.

The third rule is similar to the second one, retrieving another file from the same remote system. In this case, however, the file will be copied only when the remote file is more recent than the local copy. When the file is copied, the previous version will be retained and the file contents will be encrypted as it is transmitted across the network.

The **SplayTime** setting in the **control** section of Policy Example 10 is used to prevent the network congestion that might result from many systems simultaneously attempting to retrieve files from a master distribution host. This setting specifies a maximum number of minutes that **cfagent** will wait to begin working after it is initiated. The actual wait time is some random value less than or equal to the splay time. In this way, the working of multiple **cfagents** running with similar configurations will be sufficiently offset to avoid problems.

Another **copy** example that illustrates the use of multiple remote source systems appears in the section on feedback classes, later in this chapter.

2.1.6 Administering File Systems

The **disks** rule type allow you to verify that important file systems are present and contain sufficient free space, and the **tidy** rule type provides features for maximizing space by deleting unwanted items. Here is a sample policy using them:

Policy Example 11: Managing Disks and Disk Space

```
control:
   SensibleCount = ( 10 )

disks:  # Check disks and free space
   /homes
   /aux freespace=500mb inform=on

tidy:  # Delete core files and clear /tmp
   / include=core age=1 type=ctime
      recurse=inf xdev=off
   /tmp include=* ignore=.X11 rmdirs=sub
        age=3 recurse=inf links=traverse
```

The **disks** rules check that the */homes* and */aux* file systems are present by determining whether a reasonable number of files are present in each directory (i.e., at each mount point). The **SensibleCount** setting specifies what “reasonable” means. In the case of the latter file system, the amount of free space is also determined, and a warning is issued when the amount falls below 500 MB (including **inform=on** results in a clearer warning message).

Rules in the **tidy** stanza specify items that should be deleted from the relevant directory tree. The first rule in the example removes core dump files more than 24 hours old that are located anywhere on the disk partition containing the root directory. The second rule clears all files and empty subdirectories under */tmp* that were last accessed more than three days ago (the **rmdirs=sub** option specifies that */tmp* itself should not be removed). As **cfagent** traverses this directory tree, it will travel into subdirectories that are symbolic links to locations elsewhere in the file system (the default behavior is to skip such links).

2.1.7 Managing Processes

We now turn to a rule type that manages something other than file system objects, specifically processes. The following policies illustrate some of the capabilities of the **processes** rule type:

Policy Example 12: Managing Processes

```
processes:
   "dhcpd" matches=1
   "sendmail" restart "/etc/init.d/sendmail start"
   "inetd" signal=hup syslog=on
   "kudzu" signal=kill
   "cpuhog" matches=>2 signal=stop action=signal

   SetOptionString "-e"
```

Each rule in this **processes** stanza includes a quoted string as its initial item. This string specifies the process(es) to which the rule applies. Cfengine compares the string to the output of the **ps -ef**[16] command and applies the rule to all processes whose entry contains it. Thus, the first rule would apply to both the **inetd** and **xinetd** daemons if both were present on the system (and possibly to other processes whose **ps** output contained that string). You can modify the options supplied to **ps** via the **SetOptionString** directive.

The first rule will check for the presence of a process named **dhcpd** and issue a warning if this daemon is not running. The second rule checks for a **sendmail** process, restarting using the specified restart command it if it is not running.

The remaining rules illustrate more complex process management. The third rule will send a hangup (HUP) signal to matching processes and log its action to the **syslog** facility. The fourth rule will kill any **kudzu** process that is running. The final rule will

[16] Or **ps aux** on BSD-based systems.

count the number of **cpuhog** processes and suspend all of them when more than two are present.

There is one subtlety with respect to the **restart** option. When it is specified, it causes the corresponding command to be run if the specified process does not exist (as in the sendmail example above). It also causes the specified command to be run after an existing process is terminated via the **signal** option. This is not always what you want. More complex decision making is possible using filters, which are discussed later in this chapter.

There are also options for specifying the process's user and group execution contexts and for sandboxing: **owner**, **group**, **chdir** and **chroot**. See the *Cfengine Reference* for details.

2.1.8 Installing and Verifying Software Packages

Cfengine can also automate software package management and installation. Policies for these items are specified in the **packages** stanza. Here are some examples:

Policy Example 13: Package Management

```
control:  # Define package manager & install command
   linux:: DefaultPkgMgr = ( rpm )
   redhat:: RPMInstallCommand = ( “/usr/sbin/up2date %s” )
   suse::  RPMInstallCommand = ( “/usr/sbin/yast2 -i %s” )

packages:
   nagios version=2.4 cmp=ge
   pstree action=install
```

The settings in the **control** section specify the package management software that is in use, as well as the command used to install a software package. These directives illustrate the use of operating system–based classes within policies for defining a different installation command for different Linux distributions.

In the **packages** stanza, the first rule checks whether Nagios is installed. A warning will be generated if the package is not present or if the installed version is earlier than version 2.4. The second rule checks for the **pstree** package and installs it if it is not present on the system.

2.1.9 Executing Shell Commands

Flexible as cfengine is, there are nevertheless many actions you might want to perform that are not (yet) supported. Fortunately, cfengine provides the ability to execute arbitrary shell commands. The package itself can also be extended via modules and methods, which are discussed in detail in Chapter 5.

Before we consider policies using external shell commands, we need to look briefly at the syntax required for including quotation marks within quoted strings, since this construct is frequently needed when specifying shell commands.

> ***Definition 9: Quotes Within Quotes.*** *In cfengine you must escape quotes that are contained within quotes, even quotes of a different kind (e.g., single quotes inside double quotes). Quotes are escaped with a backslash. Here is an example of quoting within an function argument:*
>
> ```
> result = ( ExecShellResult("/usr/bin/who -r | awk \'{print $2}\'") )
> ```

Here is an example of a simple policy employing external shell scripts:

Policy Example 14: Executing External Commands

```
shellcommands:
   "/etc/init.d/postfix restart"
   "/usr/local/sbin/cleanup" timeout=300 background=true
```

The first rule runs these standard boot scripts for the Postfix facility in order to restart its associated daemons. The second rule runs a shell script named *cleanup*, located in */usr/local/sbin*. This rule starts the script as a background process, and **cfagent** will not wait for it to complete. The rule also specifies a timeout period of five minutes, after which **cfagent** will kill the corresponding process (presuming it to be hung). Be aware of the difference between this option and **expireafter**; the latter requires a second **cfagent** run in order to terminate the process.

To Shell or Not to Shell

We are used to having commands executed on a command line, where we are always working inside a shell with a defined path and environment variables. A shell can be a mixed blessing, however. For security reasons, you might want privileged scripts to avoid uncontrollable environment variables. For this reason it is possible to execute programs directly, without a shell wrapper, as in this example:

```
shellcommands:
   "/usr/bin/updatedb" useshell=false
```

The default behavior is to use a shell wrapper.

Occasionally, shell commands hang during execution due to improper file descriptor termination in children. A tip for preventing this is to explicitly close their file descriptors with this shell construction:

```
"/bin/command < /dev/null > /dev/null 2>&1" useshell=true
```

Note that you must use a shell wrapper when employing this technique.

2.2 Cfengine Classes

Classes are the cached results of tests about properties of the system. They are evaluated just before **cfagent** starts executing. Classes can be:

- Detected from the host environment.
- Defined by the system administrator in a policy file.
- Created at the start of a run based on the results of supplied functions.
- Defined as a result of actions taken (or not taken) during rule processing.
- Based on measurements/observations taken by **cfenvd**.
- Used in complex logical expressions to enable the conditional application of rules.

Classes of the first type include characteristics of the operating system environment, the network environment, and the date and time of the **cfagent** run. To see which classes are detected in your environment, run **cfagent** with the **-pv** options.

Here is an example of the output from one of our systems. We have reorganized and annotated the output for pedagogical purposes.

```
$ cfagent -pv
Defined Classes = (
bella bella_ahania_com ahania_com            Hostname & domain variations
192_168_1_101 192_168_1 192_168 19                 IP address components
April Day14 Friday Yr2006                                Date components
Hr12 Hr12_Q3 Min35 Min35_40 Q3                    Time of day components
linux linux_2_6_11_4_21_10_default                 OS and kernel version
SuSE SuSE_9 SuSE_9_3                           Operating system specifics
32_bit i686                                     Hardware characteristics
fe80__20e_35ff_fe52_5b03 net_iface_eth0      MAC address & interface name
cfengine_2 cfengine_2_1                       Cfengine version variations
UserProcs_high_dev1 DiskFree_high_dev2         System resource usage levels
...
)
```

Classes, like other identifiers, can only consist of the characters a–z, A–Z, 0–9, and the underscore. When **cfagent** converts data containing other characters such as dots or hyphens into classes, it converts all illegal characters to underscores. Hence fully qualified domain names such as *host.domain.tld*, when represented as classes, become **host_domain_tld**.

Most of these classes are self-explanatory. However, those relating to the time of day may be a bit opaque at first. First of all, these times always refer to when the current

cfagent run began, and not to the exact time when any specific rule is actually processed. The **Hr***nn* and **Min***nn* forms refer to specific hours of the day and minutes after the hour. The **Q***n* classes refer to the four 15-minute "quarters" of each hour: e.g., **Q3** refers to the period from 30 to 44 minutes after the hour. Similarly, the form **Min***mm_nn* refers to the specified five-minute interval: e.g., **Min20_25** refers to the five minutes starting at 20 minutes past the hour.

Note that, because each agent detects its own private environment, the classes it experiences are local and are not seen by any other hosts on the network.

> ***Definition 10: Autonomy and Locality.*** *By default, cfengine does not give you an overview of the state of all the hosts running it. Each host is a closed and independent box.*

If you are used to thinking in terms of centralized management, you might find this surprising or even a weakness, but how should cfengine know the boundaries of your system? No one would want a system that automatically opened every host to knowledge about every other host. Since cfengine allows every possible model from centralization to independence, it defaults to maximum privacy.

2.2.1 How Do I Make My Own Classes?

Cfengine defines a number of classes that cover generic aspects of systems. These are called hard classes because they are indisputable properties of the environment in which **cfagent** is operating. In addition to these, you might want to define other classes of your own (known as soft classes) based on abstract customizations of the local environment, such as group membership or the existence of certain files or processes.

By default, any name used as a class that is not defined in some other way/context is interpreted as a host name (or, more precisely, subject to misinterpretation as a host name).

The **classes** section of *cfagent.conf* may be used to define classes.[17] Here are some sample class definitions:

```
classes:
   WinXP = ( pc121 pc122 pc123_cfengine_org )
   indigo = ( solaris -box2 -box4 )
   TheTouched = ( FileExists(/usr/local/etc/mark) )
```

The identifier name on the left hand side becomes defined (logically true) if any of the classes on the right hand side are defined. In the first case, the class name **WinXP** is shorthand for the three specifically named hosts on the right-hand side.

In the second example, the class **indigo** is defined as an alias for any host that is of class

[17] A **classes** stanza may also be included in *cfservd.conf* for use when configuring that server.

solaris, except for host *box2* and host *box4*, where the exclusion is indicated by the minus sign.

Finally, the class **TheTouched** becomes defined if the function evaluates to true, i.e., if the file */usr/local/etc/mark* exists.

Once classes are defined, they can be used to label policy rules using the double colon notation:

```
editfiles:
 TheTouched::
    { /usr/local/etc/mark
       AppendIfNoSuchLine "Mark woz ere"
    }
```

2.2.2 Combining Classes

Classes label policy rules—promises—in the configuration files. For precise customization, you need to combine them like Boolean expressions. Classes are combined with the basic operators:

Logical Operation	Symbol	Alternative symbol
NOT	**!**	
AND	**.** (*dot*)	**&**
OR	**|**	**||**
Grouping		**( )**

Operator precedence is as ordered in this table. However, we recommend using parentheses for grouping, to avoid ambiguity as well as to improve configuration file readability.

Here are some examples of class expressions:

Class Expression	When True
solaris.Monday.Hr01::	*Solaris systems on Monday during the 1 AM hour*
aix|hpux::	*AIX or HP-UX systems*
aix.!vader::	*AIX systems other than system vader*
December.Day31.Friday::	*New Year's Eve when a Friday*
Day13.!Friday::	*The 13th of the month when not a Friday*
solaris|aix.Monday::	*Solaris systems, or AIX systems on Monday*
(solaris|aix).Monday::	*Solaris or AIX systems, on Monday*
something::	*Local hostname is "something"*

In these examples, all items that are not defined keywords are interpreted as hostnames. In the absence of user-defined and feedback classes (discussed later), this is how cfengine interprets unknown class names that it encounters.

Classes remain in effect within a stanza until another class expression is encountered. However, they do not carry across stanza boundaries. Note that the **any** class may always be used to remove any class-based restrictions in effect.

2.2.3 Defining Classes with Functions

Classes can also be defined conditionally, based on the return value of a variety of built-in functions or of an external command. Here are some examples:

```
need_restart = ( IsNewerThan(/old/file,/new/file) )
do_update = ( ChangedBefore(/etc/passwd,/1/passwd) )        Variant: AccessedBefore
has_xinetd = ( FileExists(“/usr/sbin/xinetd.conf”) )
do_import = ( IsDir(“/tmp/import”) )                       Variants: IsPlain, IsLink
ip_ok = ( IPRange(192.168.89.100-150) )
my_subnet = ( IPRange(192.168.89.200/26) )         Related: IsHost(basename,n1-n2)
is_running = ( ReturnsZero(“/bin/ps -C httpd”) )          Variant: ReturnsZeroShell
```

The functions are quite intuitive and easy to use. See the *Cfengine Tutorial* for full details on the available functions.

2.2.4 Feedback Classes

Feedback classes are classes which are defined in the course of a **cfagent** run based on whether an action took place. They are defined using the **define** and **elsedefine** options.
Here are two simple examples:

```
tidy:
   /tmp age=3 pattern=* type=mtime recurse=inf define=did_clean

disks:
   /aux freespace=500mb inform=true elsedefine=aux_ok
```

The first example defines the class **did_clean** when the **tidy** rule finds one or more files to delete. The second example defines a class named **aux_ok** when the specified file system contains at least the specified amount of free space. If **define** had been used instead of **elsedefine**, the class would be defined when there was not at least the specified amount of free space. In general, **define** creates a class when the rule results in an action, even if that action is simply issuing a warning message, and **elsedefine** creates a class when the rule results in no action.

The following example illustrates the use of feedback classes along with the **failover** option to **copy** to specify multiple potential remote source servers for a copy rule:

```
control:
   actionsequence = ( copy )
   domain = ( cfengine.org )
   AddInstallable = ( fail1 fail2 no_pam )

 copy:
 any::
    /etc/pam.d dest=/etc/pam.d recurse=1
      server=master1 failover=fail1

 fail1::
   /etc/pam.d /etc/pam.d recurse=1
      server=master2 failover=fail2

 fail2::
   /etc/pam.d /etc/pam.d recurse=1
      server=master3 failover=no_pam

alerts:
   no_pam::
      "Failed to copy PAM configuration directory."
```

This example attempts to update the PAM facility's configuration files by trying three remote servers. The **failover** option in each rule specifies a class to define if the copy operation returns an error. The structure of these rules causes three remote systems to be tried in succession. If the copy operation has still not succeeded after these three attempts, then the **no_pam** class will be defined, and the message in the **alerts** stanza will be displayed.

This example highlights two features closely tied to classes:

❖ The **AddInstallable** directive in the **control** section is used to declare feedback classes which may be defined in the course of the **cfagent** run.[18]

❖ The **alerts** stanza is used to display messages based on class expressions. The syntax is as illustrated in the preceding example: a class expression followed by a message string. Note that the **any** class may not be used, explicitly or implicitly, in the **alerts** stanza. However, you can get around this limitation by defining another class as equivalent to **any**:

```
classes:
   all = ( any )
```

[18] The **AddClasses** directive may also be used to specify classes that should be (unconditionally) activated for the current policy file. Note that we have chosen to declare all feedback classes in this guide in this way, and we recommend that you do the same. Although it is not always strictly necessary, it is a good practice.

```
alerts:
  all::
    "Hello, world."
```

2.3 Filters

Sometimes the inclusion and exclusion options do not provide sufficient flexibility to select just the items we intend. For such cases, cfengine provides filters, which can be used to build complex file and process selection expressions. A filter is a description of items that we would like to include. Filters are declared in separate stanzas in their own section of the *cfagent.conf* configuration and are attached to any number of rules, as attributes, using their identifier.

Each filter is parameterized by a number of matching criteria. Each filter has a result which is expressed as the logical combination of a number of criteria.

2.3.1 File Filter Parameters

The following components can be used to construct file filters:

- **Owner** and **Group** can use numerical IDs or names, or "none" for users or groups that are undefined in the system *passwd* and *group* files.

- **Mode** applies only to file objects. It shares syntax with the **mode=** strings in the **files** command. This test returns true if the bits that are specified as "should be set" are indeed set, and those that are specified as "should not be set" are not set.

- **Atime**, **Ctime**, **Mtime** specify times or time ranges (via the **From** and **To** prefixes—see the third example below). If the file's timestamps lie in the specified range, the expression evaluates to true. Times are specified by a six-component vector: (*year, month, day, hour, minutes, seconds*). This may be evaluated as two functions: **date()** or **tminus()** which give absolute times and times relative to the current time, respectively. In addition, the keywords **now** and **inf** (infinity) may be used.

- **Size** specifies the file's size (or a size range when the prefix **From** or **To** is included). The keyword **inf** may also be used.

- **Type** applies only to file objects. It specifies a list of file types to be matched. Items in the list should be separated by the "or" symbol, |, which make sense intuitively, since the types are mutually exclusive. Values include **reg**, **link**, **dir**, **socket**, **fifo**, **door**, and **char**.

- **NameRegex** matches the name of the file with a regular expression.

- **IsSymLinkTo** applies only when the file object is a symbolic link. It is true if the regular expression matches the contents of the link.

- **ExecProgram** matches if the command returns successfully (with return code **0**). Note that this feature introduces an implicit dependency on the command being called. This might be exploitable as a security weakness by advanced intruders.
- **ExecRegex** matches the parenthesized test string against the output of the specified command.
- **Result** is a logical expression specifying the way in which the above elements are combined into a single filter.

Here are some examples:

```
filters:
   { badgif    # Look for executables disguised as GIF
     NameRegex: “.*gif”
     ExecRegex: “/bin/file $(this) (.*ELF.*)”
     Result: “ExecRegex.NameRegex”
   }

   { histnull   # Check if users set history to dev/null
     NameRegex:  “.*history”
     IsSymLinkTo: “/dev/null”
     Result: “IsSymLinkTo.NameRegex”
     DefineClasses: “history”
   }

   { old_or_big  # Find .dat files that are old or big
     FromMtime: “date(2001,1,1,0,0,0)”
     ToMtime: “tminus(0,0,1,0,0,0)”
     FromSize: “500m”
     NameRegex: “dat$”
     Result: “NameRegex.(Mtime|FromSize)”
   }
```

In the final filter, **Mtime** is shorthand for (**FromMtime.ToMtime**), and similar abbreviations can be used for other numerical and time-period based items.

Here is another example, showing the use of a file filter in the **files** stanza:

```
filters:
  { setuid
    Owner: “root”
    Mode: “+6000”
    Result: “Owner.Mode”
  }
```

```
files:
   /home recurse=inf filter=setuid mode=-6002
    action=fixplain inform=on syslog=on
```

2.3.2 Process Filter Components

Process filter components match common fields from the ps command output:

- **PID:** process ID (parameter is a quoted regular expression).
- **PPID:** parent process ID (quoted regular expression).
- **PGID:** process group ID (quoted regular expression).
- **RSize:** resident size (quoted regular expression).
- **VSize:** virtual memory size (quoted regular expression).
- **Status:** status (quoted regular expression).
- **Command:** CMD or COMMAND field (quoted regular expression).
- **TTime:** Total elapsed time in TIME field (accumulated time). The prefixes **From** and **To** may be used to specify a range.
- **STime:** Starting time for process in STIME or START field (accumulated time). The prefixes **From** and **To** may be used to specify a range.
- **TTY:** terminal type, or none (quoted regular expression).
- **Priority:** PRI or NI field (quoted regular expression).
- **Threads:** NLWP field for SVR4 (quoted regular expression).
- **Result:** logical combination of above returned by filter (quoted regular expression).

Note that these names are all case-sensitive.

Here is a sample process filter in action:

```
filters:
# Processes owned by root with > 2 hrs CPU time
   { program_gone_bad
     Owner: "root"
     FromTTime: "accumulated(0,0,0,200,0,0)"
     ToTTime: "inf"
     Result: "Owner.TTime"
   }

processes:
   "." filter=program_gone_bad action=warn matches=0
```

In this case, the regular expression searched for among the output from **ps** is any character (specified by a single period). The **matches** option tells cfengine to perform the specified action—in this case, issuing a warning message—if any processes matching the regular expression also match the filter.

2.3.3 Troubleshooting Filters

Two common causes of error in constructing filters are:

- Incorrect capitalization of the filter attributes: e.g., "FromMTime" for **FromMtime**.
- Forgetting to include a colon after the attribute name when specifying its value:

```
FromSize "100m"
```

Will not work without the colon!

2.4 Policy Ordering and Execution

Cfengine provides several means of specifying the order in which rules are applied and actions are carried out. In the first instance, the order of processing for the various policy stanzas is controlled by the **actionsequence** directive in the **control** section, as in this example:

```
control:
   actionsequence = ( files editfiles copy )
```

In this case, the **files** stanza will be processed first, followed by the **editfiles** stanza, and then by the **copy** stanza (regardless of the order in which they appear in the policy file). Any other rule types which may be present in the policy file will be ignored.

In general, the following principles govern rule-processing order in cfengine:

- Order of rule types is guided in bulk by action sequence.
- The order of active rules is sequential within each rule type, but class dependencies can alter this, so do not assume too much.
- We can organize input into multiple files using **imports**. The ordering complexities that arise in this case are discussed in the next subsection.
- Additional ordering constraints imposed by feedback classes are taken into account.
- Two passes of the action sequence are made automatically if there remains a possibility of outstanding rules.

2.4.1 Using Multiple Policy Files

You can break up a configuration into smaller pieces, for special purposes, using the **imports** directive. If you use this feature, then the recommended practice is to use the *cfagent.conf* file only to import other files. The reason for this is straightforward. Cfengine reads imported files after the *cfagent.conf* file, so any definitions you put into **imports** will not be seen by configuration commands that you place in *cfagent.conf*. If you use *cfagent.conf* only for imports, then you can first import definitions, then commands that depend on the definitions, and so on.

Here is a sample *cfagent.conf* file broken down into several components:

```
imports:
  any::
    cf.main
    cf.classes
    cf.motd
    cf.services
    cf.mail

 # OS-specific policies
  solaris:: cf.solaris
  linux:: cf.linux
  redhat:: cf.redhat
  SuSE:: cf.suse
  nt:: cf.windows

# More specific policy files
  (linux|solaris).!matrix:: cf.users
  gaughin|vangogh|picasso|okeefe|matisse:: cf.printservers
  nexus|dax|cube|pax:: cf.www
  matrix:: cf.matrix
```

Each of these files is stored in */var/cfengine/inputs*. Note that the policies may be divided into separate files on any basis that makes sense at your site. In this example, we have divided rules based on both operating system and functional area.

Note that we use only hard classes in this top-level policy file. We also place policy files applying to only a few hosts near the end of the file, moving from most general to most specific.

Each of the individual included files can contain a **control** section, if needed, to define lists, variables, etc., used in that file. However, only the first **actionsequence** directive encountered will be used; any ones contained in later files will be ignored. Because of this rule, many people choose to place the **actionsequence** directive in the top-level file.

3. Building a Cfengine Infrastructure

In this chapter, we provide a quick setup roadmap for distributed management using cfengine. For this, you will need the server component **cfservd** running on at least one master host, as well as the agent running on each of your slave hosts and master host.

The traditional view of network management is to apply a control over a network from some centralized, authoritative location: a master host. You can easily create this kind of architecture using cfengine, but you are not limited by it. Cfengine's principle of autonomy makes it plausible to divide authority into regions, or have every host managed individually if that suits your needs. There is no compulsion to have centralized management, but it is easily implemented if that's what makes sense for you.

What are the advantages of centralized management?

- ❖ Having a single point of decision aids consistency.
- ❖ Changes of policy are easiest to implement from one place.
- ❖ Backup and version control of policy are convenient when policy is centralized.

What are the disadvantages?

- ❖ Central services are somewhat old-fashioned, making one think of marching armies rather than free market business.
- ❖ Local customization becomes awkward, by forcing local knowledge to pass upward through a central authority.
- ❖ Centralization is inappropriate for security and privacy if you have completely independent departments or businesses that merely coexist and work together.

You can probably think of other reasons, and indeed you should think about this carefully. The key to sound management lies in calculating the correct force to apply. Too much, and you will bludgeon your departments into inappropriate conformity; too little, and they might run away to a place you no longer understand.

Cfengine's view is one of voluntary cooperation and hence voluntary consensus.

3.1 Roadmap for Centralized Policy

We shall assume that you have a central location for your policy. If you don't, then you can repeat this procedure multiple times for each decentralized point of control.

There are several steps to be performed. The following order is recommended:

1. Set up the policy source host first. This is the computer that will store the master policy files which are used on every host running cfengine.
   a. Decide on the policy for the network.
   b. Make production versions of *cfagent.conf* and *update.conf.* The first file is the primary policy file to be applied to every host. Your set up may eventually include a series of policy files rather than just a single one, but this default policy file is a good way to get started. The second file, *update.conf,* is the policy file used by clients to retrieve updated versions of the configuration files from the policy host whenever they change. In other words, it is the policy file that enables cfengine to update itself.
2. Set up clients to install and update themselves.
   a. Install an appropriate version of the software binaries on each host. As part of the installation process, you should add an entry for cfengine to */etc/services*, specifying port 5308/tcp.
   b. Make a small, generic *cfagent.conf* bootstrap file to start the automation process rolling. This file will be used for the first policy file retrieval from the policy host. Thereafter, the real policy files will be present on and used by the client. This file is also used to update the cfengine binaries when needed.

Deciding policy is clearly a big task, but you can start simply, by getting a small policy running across your network and then building on this foundation. To get started, you only need to have a working prototype.

3.1.1 Set Up the Policy Host Server

The best place to start is at the computer that will house your policy and be used to distribute it to the rest of the network. Once this host is set up, all of the others can use it to bootstrap.

Procedure 5: Set Up the Policy Host

1. Choose a host to play the role of policy master.
2. Set up the **cfservd** component for distributing policy to authorized hosts.
3. Create your prototype master configuration files (probably as stubs to be developed later) in a separate directory tree (e.g., */master/cfengine*). The cfengine command binaries may also be placed within this tree in order to update them using cfengine.
4. Grant access to the master files to your client hosts in *cfservd.conf.*
5. Create and install the policy files on the master. Don't forget **cfservd**.

Your *cfservd.conf* file can be located in a master repository. We shall use the path */master/cfengine/inputs* to designate this. You can replace that with your own location. We will also assume that you are managing an IP subnet, with address series of the form 192.168.0.*nnn*. Finally, we'll call our example master host *polly.cfengine.org* (although no such system exists).

In this example, we are configuring several layers of access. The **cfservd** daemon independently controls the following activities:

- The right to connect to the server daemon: **AllowAccessFrom**.
- The right to make multiple requests of the server: **AllowMultipleConnectionsFrom**.
- The right to present new credentials (public encryption key) on trust: **TrustKeysFrom**.
- The right to read files from the server's disk: **admit** rules.

Here is a sample *cfservd.conf* configuration file which you can use as a model:

Policy Example 15: An Initial cfservd.conf File

```
# Basic cfservd.conf (improve later)
control:
   domain = ( cfengine.org )
   AllowAccessFrom = ( 192.168.0 )
   AllowMultipleConnectionsFrom = (192.168.0 )
   TrustKeysFrom = ( 192.168.0 )

classes:
   policyhost = ( polly ) # Alias for polly.cfengine.org

admit: # provides read access to files on the server
  policyhost::
   /master/cfengine
      192.168.0      # By IP subnet
      *.cfengine.org   # By domain
```

Note that this file does not yet reside in a location where **cfservd** can find it. This will be handled by the bootstrap *update.conf* (discussed below).

The **TrustKeysFrom** directive deserves special mention. This directive should be activated only long enough to allow initial key exchanges from new client computers, a process which should be monitored carefully. Afterward, it should be commented out. The cfengine key infrastructure and the issues it raises are discussed later in this chapter.

3.1.2 Set Up the Master Policy Files

The next step is to create two policy files, *cfagent.conf* and *update.conf*, the master policy file and policy file updater, respectively. As we've seen, *cfagent.conf* is the default policy file processed by **cfagent**; *update.conf* is another file that has special meaning to **cfagent** and it is automatically processed at the start of each run. We will create these files in the directory */master/cfengine/inputs* on the policy server.

Your starting master policy file can be rather simple. In our example, all it does is to install a **cron** job to run cfengine every 15 minutes and email the output to *sysadmin@cfengine.org*.

Policy Example 16: An Initial Master Policy File

```
# Prototype cfagent.conf — add to/customize over time
control:
   actionsequence = ( editfiles processes )
   EmailTo = ( syadmin@cfengine.org )
   bin = ( /var/cfengine/bin )
   access = ( root mark aeleen )  # Who is allowed to run cfagent

editfiles:
  !SuSE::
   { /var/spool/cron/crontabs/root
      AutoCreate
      AppendIfNoSuchLine
        "0,15,30,45 * * * * ${bin}/cfexecd -F"
   }

processes:
# Activate these lines to start these daemons
#    "cfservd" restart "/var/cfengine/bin/cfservd"
#    "cfenvd" restart "/var/cfengine/bin/cfenvd"
```

We've omitted the policy for SuSE Linux systems this time.

In order to complete the infrastructure setup, you will need to install a policy file on the policy master computer (e.g., */var/cfengine/inputs/cfagent.conf*) . Policy Example 16 can serve as a starting point for that file as well. Just be sure to activate the rule for **cfservd**.

We now turn to the update policy file. This file can handle updating of the *cfagent.conf* master policy file, the cfengine binaries, and itself. Here is our initial version:

Policy Example 17: Starting Update Policy File

```
# Prototype update.conf — customize as appropriate
control:
    actionsequence = ( copy processes )
    domain = ( cfengine.org )
    policyhost = ( polly )
    master_cfinput = ( /master/cfengine/inputs )
    master_modules = ( /master/cfengine/modules )
    cfdir = ( /var/cfengine )
    binsrc = ( /master/cfengine/bin )

    SplayTime = ( 2 )
    AddInstallable = ( newservd )

copy:
    ${master_cfinput} server=${policyhost}
      dest=${cfdir}/inputs
      recurse=inf mode=700
      type=binary trustkey=true

    ${master_modules} server=${policyhost}
      dest=${cfdir}/modules recurse=1
      mode=700 type=binary

    ${binsrc}/cfagent server=${policyhost}
      dest=${cfdir}/bin/cfagent
      backup=false type=checksum

    ${binsrc}/cfservd server=${policyhost}
      dest=${cfdir}/bin/cfservd
      backup=false type=checksum
      define=newservd

    ${binsrc}/cfexecd server=${policyhost}
      dest=${cfdir}/bin/cfexecd
      backup=false type=checksum

processes:
  newservd::
    "cfservd" signal=term restart "${cfdir}/bin/cfservd"
```

Note that we are including an optional update of three of the cfengine binaries from a central host. You might prefer to handle software updates differently. A similar directive can be added to the **processes** stanza for **cfenvd** if you so choose.

You will also want to create a simple *cfservd.conf* file for use on client systems, placing it in the same master location. For most clients, it can be very simple, consisting only of a **domain** directive in the **control** section (see Policy Example 15).

3.1.3 Set Up a Bootstrap File on Each Client

The final step is to create a bootstrap *cfagent.conf* policy file for client systems that will be distributed along with the **cfagent** binary, perhaps as part of the operating system installation. This file retrieves the policy files from the master policy host. It can also run **cfkey** and/or start the cfengine daemons, if desired.

Policy Example 18: Example Client Bootstrap Policy File

```
# This is a stub that will be overwritten later
control:
   actionsequence = ( shellcommands copy processes )
   domain = ( cfengine.org )
   cfdir = ( /var/cfengine )
   policyhost = ( polly )
   master_cfinput = ( /master/cfengine/inputs )

copy:
   ${master_cfinput} server=${policyhost}
     dest=${cfdir}/inputs recurse=inf
     mode=700 type=binary trustkey=true

shellcommands:
#  Activate this line if cfkey was not run during cfengine installation.
#  /usr/local/sbin/cfkey

processes:
#  Activate these lines to start these daemons
#  “cfservd” restart “/var/cfengine/bin/cfservd”
#  “cfenvd” restart “/var/cfengine/bin/cfenvd”
```

The first time **cfagent** runs, this file copies the master cfengine *inputs* directory to the local system. The next time **cfagent** runs, the *update.conf* policy is executed (since it is now present on the local system), updating the cfengine configuration (and potentially the software itself), and then the master version of *cfagent.conf* installs cfengine as a **cron** job, running every 15 minutes.

3.1.4 The cf.preconf File

Cfengine provides another hook for getting systems running. The file *cf.preconf* is executed before each run. This file, which is optional, is a shell script (written in any language) that is designed to get the system minimally configured so that cfengine can run. It can be useful for initial system setup and also when the system is in a very bad way for some reason.

Here is a simple version of this file, written as a Bourne shell script:

Policy Example 19: Sample cf.preconf File

```
#!/bin/sh

backupdir=/nexus/master/etc

if [ ! -s /etc/resolv.conf ]; then
  echo Creating minimal resolv.conf file
  cat > /etc/resolv.conf << XX
domain cfengine.org
nameserver 192.168.0.100
nameserver 192.168.0.101
XX
fi

if [ ! -s “/etc/passwd” ]; then
  echo Replacing missing passwd file
  if [ “$1” == “freebsd” ]; then
    /bin/cp $backupdir/passwd.bsd /etc/passwd
  else
    /bin/cp $backupdir/passwd /etc/passwd
  fi
fi
```

Similar code for other key files: shadow, group, nsswitch.conf

3.1.5 Policy File Summary

The following summarizes the configuration files present on each system at various points in the cfengine life cycle:

- Policy host, normal operation: Cfengine is set up and running on this system. Master versions of policy files, including at least *cfagent.conf, cfservd.conf,* and *update.conf,* are present in */master/cfengine/inputs*. Similarly, cfengine binaries are present in */master/cfengine/bin,* and other subdirectories of */master/cfengine* are populated if appropriate (e.g., *modules*). The *cfservd.conf* file for this host must also allow access from the various client systems.

- Client system, initial setup: Cfengine software is installed, including the bootstrap version of */var/cfengine/inputs/cfagent.conf.* The latter can be installed with cfengine or copied to the client system manually.

- Client system, normal operation: Copies of the policy files and **cfservd** configuration file from the master policy host are present in */var/cfengine/input*. These files and the cfengine command binaries are updated as necessary by *update.conf*.

3.2 The cfrun Command: Simulating Push with Pull

Cfengine forbids directly pushing data and files in any of its rules. Nevertheless, the software does provide a method for a user to request a running **cfservd** server to run **cfagent**. The user cannot send the policy that is to be executed, but can only suggest to the listening server that it run the policy it already has.
If the policy includes checking for updates from an external location, then this is a way of simulating a push of a new version of your configuration policy to a remote host.

The **cfrun** program is a simple tool that polls hosts in a list and sends the relevant request to each host in turn. It can also parallelize its operation, to speed up matters.

3.2.1 Configuring cfservd to Listen

Client systems must be preconfigured to grant access to remote users initiating **cfagent** runs via **cfrun**. You must declare the name of the command that will be executed in **cfservd**'s configuration file, as in Policy Example 20.

Policy Example 20: Enabling Remote Command Initiation

```
control:
   AllowUsers = ( root mark aeleen )
   cfrunCommand = ( "/var/cfengine/bin/cfagent" )
#    cfrunCommand = ( "/var/cfengine/bin/cfexecd -F" )

classes:
   hostlist = ( IPRange(192.168.10.0/24) )

admit:
   /var/cfengine/bin/cfagent hostlist
#    /var/cfengine/bin/cfexecd hostlist
```

There are two ways you might want to set this up. In the first case, which is active in Policy Example 20, this feature is configured for interactive use. By running **cfagent** directly, the output will be channeled back to you immediately. In the second case, which is commented out in the example, **cfagent** will be run under the **cfexecd** wrapper and any output will be emailed to the usual recipient.

3.2.2 The cfrun.hosts File

To use **cfrun**, you start by creating a configuration file called *cfrun.hosts* in */var/cfengine/inputs*. In its simplest form, it contains a list of host names (one per line) that are to be contacted. If there are special options you would like to set, globally or for a particular host, you can place them in the file as well. For example, the domain name is needed if you are going to use unqualified host names and your name resolution mechanism does not append a default domain.

Here is a sample file:

```
domain=cfengine.org
access=mark,aeleen
outputdir=/tmp/cfoutput                          Directory must already exist!
maxchild=20
hostnamekeys=true

gudinne
wallace:3333
...
include=cfrun.external.hosts
```

Here, we define a list of allowed users, a directory for placing the output of **cfrun** queries, and a maximum number of children to spawn for parallel connections, and we require that hosts use key-based authentication. We also list two hosts (one with an alternate communication port), as well as incorporating an external file containing still more hosts, via the **include** directive.

If the **outputdir** directive is in place, **cfrun** automatically uses a parallelized batch method of communication.

3.2.3 Using the cfrun Command

The **cfrun** command itself has the following form:

```
cfrun - local options [host list] \-\-  remote options \-\-  remote classes
```

If no host list is included, then all of the hosts in *cfrun.hosts* are contacted.

Since **cfrun** addresses remote hosts from a local host, there is an ambiguity in whether options are intended for the **cfrun** command itself or are meant to be passed on to the agent on the remote hosts. To clarify this distinction, the arguments are organized as follows:

- Local options are processed by **cfrun** on the local host.
- Remote options are passed on as options to the remote **cfagent** (actually, to the command defined in **cfrunCommand**). Note, however, that it is not possible to send the **-f** option to the remote agent to ask it to run a different policy file. This option is stripped by the server on receipt, to prevent an unauthorized attempt to change policy.

- ❖ Remote classes are processed by the remote **cfservd** service; they specify classes that must be satisfied by the remote host in order to invoke the remote command.

Here are some examples, all of which use the host list in *cfrun.hosts*:

cfrun -- -- linux	*Run on all machines, specifying class linux.*
cfrun -- -p	*Just parse policy file on all hosts.*
cfrun -v -- -v	*Verbose output on local system and remote hosts.*
cfrun -v -- -k -- solaris	*Verbose local; suppress copy/specify solaris on remote.*

Table 3.1 lists the most useful options to the **cfagent** command.

Option	**Effect**
-c	Ignore the **files** section.
-d	Produce more output for debugging purposes.
-D *classes*	Define/activate the specified classes for this run.
-e	Ignore the **editfiles** section.
-f *file*	Run with the specified policy file (invalid via **cfrun**).
-H	Deactivate all hard classes.
-i	Ignore the **interfaces** section.
-I	Make inform the default for rules.
-k	Ignore the **copy** section.
-l	Set the default to traverse links in recursive directory operations.
-L	Set the default to delete stale links.
-M	Ignore modules.
-n	Display actions that would be taken, but don't do anything.
-N *classes*	Undefine/deactivate the specified classes for this run.
-p	Just parse the policy file and then exit.
-pv	Display list of hard classes.
-s	Ignore the **shellcommands** section.
-t	Ignore the **tidy** section.
-v	Display verbose output messages.
-w	Suppress warning messages.
-x	Ignore the **links** section.

Table 3.1 Options to the cfagent Command

3.3 DHCP and Dynamic Addresses

If you are using dynamic addressing for hosts, cfengine will struggle to justify its trust in the public keys it sees. To make sense of public keys, one requires both a key and an independent identity to tie it to. Once a key has been trusted, the key alone is (in principle) sufficient identity (just as a fingerprint is good enough to represent you, once it has been registered).

When IP addresses change, **cfservd** loses its coupling between address and key identity, and so it has to start reevaluating. There are two cases to consider. The default behavior is as follows:

- A host with a known IP address presents a new key to **cfservd** because, for instance, the key was changed. In this case **cfservd** complains that the keys do not match and refuses to authenticate the host.
- A host with an unknown IP address presents a known key to **cfservd**. In this case, cfengine will look for a key file bound to the new address and will not find one, so **cfservd** treats the key as unknown and normal trust rules for accepting a new key apply.

This can cause a problem for hosts with IPv4 addresses given by DHCP, since the address-key binding is only temporary. **cfservd** works around this by allowing you to define a new access list called `DynamicAddresses = ( 192.240.1 )`.

If a connecting address is unknown but lies in this range, **cfservd** will look in a database to see if the key itself is known, bound to a different address. If the key is found, **cfservd** trusts the binding and re-binds the key to the new IP address. If the key is not known, then (again) normal trust rules apply for accepting a new key.

Servers should always have fixed IP addresses in general; otherwise you have no idea to whom you are connecting.

3.3.1 Public Key Exchange Issues

Public key exchange is a subtle idea. The issues are not difficult to understand, but they are seldom expressed very clearly.

> ***Definition 11: Public key exchange.*** *The principle of a public key system is to associate an identity with a key, which can be freely made public knowledge. The difficulty is not in distributing the key, but in being certain who is really the owner of a key that you have received.*

Imagine that you wanted to know the identity of a stranger whom you have never seen before. What proof would you accept of his or her identity? A name tag? A letter of reference? A DNA test? Of course, an evil identical twin would be able to pass all of these tests and still fool you.

It is important to understand that trust in identity is something we always grant in an entirely personal way. There is always a chance of being wrong: cryptographic methods do not eliminate this doubt (although they are sometimes presented as doing so). Trust is risk, and you simply have to live with it.

There are two common models of trust management:

- The trusted third party: This method is used by the Web. Here you outsource the trust decision to a third party who claims to know whether keys are genuine. This provides centralized management of trust. Since cfengine is about decentralization, this method is not used by cfengine.
- The challenge: Individual decision-making is the way to make trust decisions for autonomous agents. They want to rely only on their own judgment. The secure shell uses this approach. Instead of asking you to trust an authority/third party, the software requires you to assume the risk, and you must decide whether to accept offered keys.

Key exchange in cfengine is very similar to that in the secure shell. However, cfengine uses separate keys from ssh that are generated using the **cfkey** program.

Cfagent is not an interactive program, so when it first receives a key, the decision as to whether to accept it must be made non-interactively. The natural solution is to make this decision part of policy. There are several methods for accomplishing this:

- Client side: You can add **trustkey=on** to a remote copy command to accept the named remote-server's key.
- Client side: You can distribute the keys collected by one client to others by copying the */var/cfengine/ppkeys* subdirectory.
- Server side: The server can receive keys from many sources; the access control list for this is the **TrustKeysFrom** directive. If a server IP address is in this list, it will be accepted.

Once a key has been accepted on trust, it is cached locally and used to verify subsequent connections, until you revoke the key by explicitly deleting the key from */var/cfengine/ppkeys*. This functioning is very similar to how the secure shell handles key exchange and trust.

Once the key has been exchanged, the **trustkey** option has no effect. Thus, on the client side it does not matter if you leave **trustkey** options in place. On the server side, however, remember that the trust rule potentially applies to many hosts, and you might not have considered the full ramifications of your setting. Always take care with trust issues. The best practice on the server side is to be as specific as possible, limiting the **TrustKeysFrom** list to a small set of explicitly named hosts.

3.4 Dealing with Firewalls

Some users want to use cfengine's remote copying mechanism through a firewall, in particular to update the cfengine policy on hosts inside a DMZ (the so-called de-militarized zone). Firewalls are often shrouded in myth and mystery: magical force fields that protect us against Klingon torpedoes. It is important to see the firewall security model together with the cfengine security model. Among the difficulties one faces is the fact that the firewall administrator is not often the same as the cfengine administrator and does not trust anyone or anything. You might have to convince this person to make changes that help you out, so it is important to understand the consequences of your security strategy.

Any piece of software that traverses a firewall can, in principle, weaken the security of the barrier. On the other hand, a strong piece of software might have better security than the firewall itself. Consider the example in Figure 3.1.

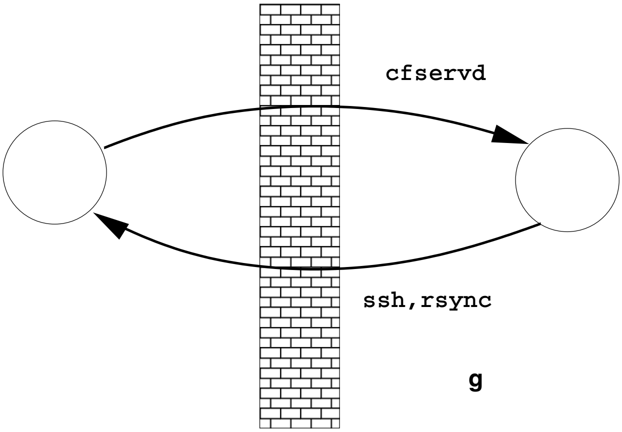

Figure 3.1: A Cfengine Host Outside a Firewall

For convenience, we label the regions inside and outside the firewall as the “secure area” and “demilitarized zone.” It should be understood that the area inside a firewall is not necessarily secure in any sense of the word unless the firewall configuration is understood together with all other security measures.

Our problem is to copy files from the “secure” source machine to hosts in the DMZ, in order to send them their configuration policy updates. There are two ways of getting files through the firewall:

- An automated cfengine solution, i.e., a pull operation generated outside with a target inside the secure area.
- A manual push to the outside of the wall from the inside.

One of the main aims of a firewall is to prevent hosts outside the secure area from opening connections to hosts in the secure area. If we want **cfagent** processes on the outside of the firewall to receive updated policies from the inside of the firewall, information has to traverse the firewall.

> ***Definition 12: Conflicting Trust Models.*** *Cfengine's trust model is fundamentally at odds with the external firewall concept. Cfengine says: "I am my own boss. I don't trust anyone to push me data." The firewall says: "I only trust things that are behind me." The firewall thinks it is being secure if it pushes data from behind itself to the DMZ. Cfengine thinks it is being secure if it makes the decision to pull the data autonomously, without any orders from some potentially unknown machine. One of these mechanisms has to give if firewalls are to co-exist with cfengine.*

From the firewall's viewpoint, push and pull are different: a push requires only an outgoing connection from a trusted source to an untrusted destination; a pull necessarily requires an untrusted connection being opened to a trusted server within the secure area. For some firewall administrators, the latter is simply unacceptable (because they are conditioned to trust their firewall). But it is important to evaluate the actual risk. We have a few observations on this score to offer:

- It is not the aim here to advocate any one method of update. You must decide for yourself. The aim is only to evaluate the security implications. Exporting data from the secure area to the DMZ automatically downgrades the privacy of the information.
- The cfengine security model assumes that the security of every host will be taken seriously. A firewall should never be used as a substitute for host security.
- Knowing about cfengine but not your firewall or your secure network, it is only possible to say here that it seems, to us, safe to open a hole in a firewall to download data from a host of our choice, but we would not accept data from just any host on your company network on trust. It would be ludicrous to suggest that an arbitrary employee's machine is more secure than an inaccessible host in the DMZ.

3.4.1 Option: A Policy Mirror in the DMZ

You can compromise by creating a policy mirror in the DMZ. This is the recommended way to copy files, so that normal cfengine pull methods can then be used by all other hosts in the DMZ, using the mirror as their source. The policy mirror host should be as secure as possible, with preferably few or no other services running that might allow an attacker to compromise it. In this configuration, you are using the mirror host as an envoi of the secure region in the DMZ.

Any method of pushing a new version of policy can be chosen in principle: CVS, FTP, RSYNC, SCP. The security disadvantage of the push method is that it opens a port on the policy mirror, and therefore the same vulnerability is now present on the mirror, except that now you have to trust the security of another piece of software as well. Since this is not a cfengine port, no guarantees can be made about what access attackers will get to the mirror host.

3.4.2 Option: Pulling Through a Wormhole

Suppose you are allowed to open a hole in your firewall to a single policy host on the inside. To distribute files to hosts that are outside the firewall it is only necessary to open a single tunnel through the firewall from the policy mirror to the cfengine service port on the source machine. Connections from any other host will still be denied by the firewall. This minimizes the risk of any problems caused by attackers.

To open a tunnel through the firewall, you need to alter the filter rules. A firewall blocks access at the network level. Configuring the opening of a single port is straightforward. We present some sample rules below, but make sure you seek the guidance of an expert if necessary.

Cisco IOS rules look like this:

```
ip access-group 100 in
access-list   100 permit tcp mirror host source eq 5308
access-list   100 deny  ip any any
```

Linux iptables rules might look something like this:

```
iptables -N newchain
iptables -A newchain -p tcp -s mirror-ip 5308 -j ACCEPT
iptables -A newchain -j DENY
```

Once a new copy of the policy is downloaded by cfengine to the policy mirror, other clients in the DMZ can download that copy from the mirror. The security of other hosts in the DMZ is dependent on the security of the policy mirror.

3.4.3 Frequently Asked Questions About the Pull Method

- ❖ *Doesn't opening a port on a machine on the inside of the firewall make it vulnerable to both denial-of-service and buffer overflow attacks?*

  Buffer overflow attacks are extremely unlikely in cfengine by design. The likelihood of a bug in cfengine should be compared to the likelihood of a bug existing in the firewall itself.

  Denial-of-service attacks can be mitigated by careful configuration (see separate FAQ item). **cfservd** reads a fixed number of bytes from the input stream before deciding whether to drop a connection from a remote host, so it is not possible to mount a buffer overflow attack before an invalid host IP is rejected.

  Another possibility is to use a standard VPN to the inside of the firewall. That way one is concerned first and foremost with the vulnerabilities of the VPN software.

- *Doesn't opening the firewall compromise the integrity of the policy information by allowing an attacker a chance to alter it?*

  The cfengine security model, as well as the design of the server, disallows the uploading of information. No message sent over the cfengine channel can alter data on the server. (This assumes that buffer overflows are impossible.)

- *Couldn't an IP spoofer manage to gain access to data from the policy server that it should not be able to access?*

  Assuming that buffer overflow attacks and DOS attacks are highly improbable, the main worry with opening a port is that intruders will be able to gain access to unauthorized data. If the firewall is configured to open only connections from the policy mirror, then an attacker must spoof the IP of the policy attacker. This requires access to another host in the DMZ and is non-trivial. However, if the attacker succeeds, the worst he/she can do is to download information that is available to the policy mirror. But that information is already available in the DMZ, since the data has been exported as part of the policy; thus, there is no breach of security. (Security must be understood to be a breach of the terms of predefined policy.)

- *What happens if the policy mirror is invaded by an attacker?*

  An attacker who gains root access to the mirror will be able to affect the policy distributed to any host in the DMZ. The policy mirror has no access to alter any information on the policy source host. Note that this is consistent with the firewall security model of trusted/untrusted regions. The firewall does not mitigate the responsibility of securing every host in a network regardless of which side of the firewall it is connected to.

Unfortunately, these decisions are often made as a matter of principle rather than considered judgment.

4. Some Case Studies: Sample Policies

In this chapter we present a gallery of sample configurations for different scenarios. They are presented "as is," with comments for your convenience. You are encouraged to browse and ponder them.

4.1 Managing a Laptop Computer

Even laptop computers can benefit by running cfengine. Configuration management and maintenance applies to any computer in a work environment. Two issues with laptops are: configuration problems with wireless networks, and backups. Since **cfagent** cannot push files to other computers due to its strict security policy, we cannot make a backup to a foreign medium (in the absence of mounted remote file systems). Nevertheless, even a local copy of files is a good idea, especially when you are far away from any network. If you accidentally do the **rm -r** thing and delete a bunch of files, it is comforting to know that you have a fairly recent copy on your machine. Thus, Mark's laptop configuration contains an automated copy of files to the same disk.

Here is the **control** section. Recall that the **netconfig** item in the **actionsequence** is the **actionsequence** item for the various network interfaces-related rule types.

```
# Example laptop policy
control:
   actionsequence =
    ( netconfig resolve copy shellcommands editfiles )

   domain = ( cfengine.org )
   sysadm = ( cfengine@cfengine.org )
   nosey = ( root@cfengine.org )
   smtpserver = ( smtp.cfengine.org )
   ntpserver = ( ntp.cfengine.org )

   timezone = ( MET CET ) # Welcome to Norway ...
```

The next section of the file configures the network interface when necessary. The **resolve** section illustrates the use of a literal line for the */etc/resolv.conf* file by including the desired text within quotation marks. This stanza also employs the hard class corresponding to the laptop's subnet.

```
defaultroute:
  no_default_route::
    192.168.1.254

resolve:
# Expect laptops to use DHCP
  "search cfengine.org"

192_168_1::
    192.168.1.254 # NAT gateway
    192.168.20.87
```

Next comes the **copy** stanza, which specifies the local backup operation mentioned earlier:

```
copy:
# Do a backup to a local directory in case of accidents
    /home/mark dest=/home/backup
    recurse=inf ifelapsed=240
```

The **editfiles** rules ensure that SSH is configured properly. This example illustrates the use of some additional editing operations.

```
editfiles:
    { /etc/ssh/sshd_config  # SSH daemon
      AppendIfNoSuchLine "Banner /etc/ssh/banner"
      ReplaceAll "X11Forwarding.*no"
        With "X11Forwarding yes"
      AppendIfNoSuchLine "X11Forwarding yes"
      ReplaceAll "PrintMotd.*yes"
        With "PrintMotd no"
      AppendIfNoSuchLine "PrintMotd no"
      DefineClasses "sshd_hup"
    }

    { /etc/ssh/ssh_config  # SSH client
      ReplaceAll "ForwardX11.*no"
        With "ForwardX11 yes"
      AppendIfNoSuchLine "ForwardX11 yes"
      }
```

The final stanza is **shellcommands**, which synchronizes the local time with an NTP server and also works around a Linux misfeature in which the wireless network interface's device name can change from boot to boot.

```
shellcommands:
   "/usr/sbin/ntpdate $(ntpserver) > /dev/null"

# Stupid wireless changes the device all the time
# Run: cfagent -Dfix_net
# One of these will fail, but who cares?
# The other will pin our wandering wireless signal ...
 fix_net::
   "/usr/sbin/iwconfig eth1 essid MyHomeNet"
   "/usr/sbin/iwconfig eth0 essid MyHomeNet"
```

This configuration is small and simple—the best kind!

4.2 Web Server

In a production environment, we would generally like to have systems completely automated with no hands-on work at all. This requires some infrastructure to get going.

The aim of this policy file is to take a computer box straight from installation (e.g., by CD-ROM or DVD) and then not make any changes by hand. Every change has been automated from the start. This example sets up a SuSE Linux machine with a Web server, PHP, and a Subversion version control system repository.

Here are the **control** and **classes** sections of the policy file:

```
# Web server policy
control:
   domain = ( cfengine.org )
   smtpserver = ( smtp.cfengine.org )
   EmailTo = ( cfengine@cfengine.org )
   IfElapsed  = ( 20 )
   ExpireAfter = ( 240 )
   EditfileSize = ( 40000 )
   actionsequence =
     ( directories editfiles files copy
       links shellcommands processes )

#  Define macros
   repositories = ( /files/projects )
   masterfiles = ( /master )
   htdocs = ( /files/htdocs )

#  Class/command to check Subversion database availability
   AddInstallable = ( viewcvs_error )
   probewww = ( ReadTCP("project.cfengine.org",
     "80","GET /viewcvs HTTP/1.0 ${n}${n}","512") )
```

```
classes:  # Check probe output for error condition
   viewcvs_error =
     ( RegCmp(".*Exception Has Occurred.*","${probewww}") )
```

The policy file illustrates defining a class when an error condition is detected.

Here are the **directories** and **files** rules. The former is a special version of **files** that creates specified directories if they do not exist:

```
directories:
   ${repositories} mode=755
   ${htdocs} mode=755
   ${repositories}/Cfengine-2
   ${repositories}/Archipelago
   ${repositories}/LabCompendium
   ${repositories}/UNIXCompendium

files:
   ${repositories} recurse=inf owner=wwwrun
     action=fixall ifelapsed=5

   ${htdocs} recurse=inf mode=0644 action=fixall
     ignore=*wiki* ignore=consortium

   ${htdocs}/consortium recurse=inf mode=a+rw
     owner=root action=fixall

   ${htdocs}/mediawiki-1.4 recurse=inf mode=a+r
     owner=root action=fixall ignore=config ignore=images
```

Many more file specifications

Here are the **links** and **copy** stanzas, which install key files needed by the various facilities on the Web server system:

```
links:
   /srv/www/htdocs -> /ourfiles/htdocs
   /ourfiles/htdocs/wiki -> ./mediawiki-1.4

copy:
  # ViewCVS config template, edited by hand and placed in masterfiles
   ${masterfiles}/VIEWCVS_styles.css
     dest=/srv/viewcvs/doc/styles.css mode=0644
   ${masterfiles}/viewcvs.conf
     dest=/srv/viewcvs/viewcvs.conf mode=0644
```

```
# Install key files
    ${masterfiles}/server.key dest=/etc/apache2/ssl.key/server.key
      mode=400 type=checksum
    ${masterfiles}/server.csr dest=/etc/apache2/ssl.csr/server.scr
      mode=400 type=checksum
    ${masterfiles}/server.crt dest=/etc/apache2/ssl.crt/server.crt
      mode=400 type=checksum
    ${masterfiles}/ca.key dest=/etc/apache2/ssl.key/ca.key
      mode=400 type=checksum
    ${masterfiles}/ca.crt dest=/etc/apache2/ssl.crt/ca.crt
      mode=400 type=checksum

# Wiki setup
    ${masterfiles}/Wiki-LocalSettings.php
      dest=/files/htdocs/wiki/LocalSettings.php
      mode=644 type=checksum
```

Next, we edit some of the local configuration files. Some files can be copied from a template; others are best edited in a more intelligent way, because operating system updates might include things that we would miss out on if we simply overwrote the entire file.

```
editfiles:
    { /var/spool/cron/tabs/root
      AutoCreate
      AppendIfNoSuchLine
        “0,15,30,45 * * * * /var/cfengine/bin/cfexecd -F”
    }

    { /etc/sysconfig/apache2
      BeginGroupIfNoLineMatching
        “APACHE_SERVER_FLAGS=\”SVN_VIEWCVS\” ”
        ReplaceAll
          “APACHE_SERVER_FLAGS=\”\”\
        With
          “APACHE_SERVER_FLAGS=\”SVN_VIEWCVS\” ”
      EndGroup

      BeginGroupIfNoLineMatching
        “APACHE_CONF_INCLUDE_FILES=\”/master/my-http.conf\” ”
        ReplaceAll
          “APACHE_CONF_INCLUDE_FILES=\”\” ”
        With
          “APACHE_CONF_INCLUDE_FILES=\”/master/my-http.conf\” ”
      EndGroup
    BeginGroupIfNoLineMatching ".*php4 dav dav_svn.*"
      ReplaceAll "php4" With "php4 dav dav_svn"
    EndGroup
 }

  { /etc/postfix/main.cf
    ReplaceAll "^mydomain =.*" With "mydomain = iu.hio.no"
    ReplaceAll "^relayhost =.*"
    With "relayhost = [nexus.iu.hio.no]"
    AppendIfNoSuchLine "relayhost = [nexus.iu.hio.no]"
    AppendIfNoSuchLine "mydomain = iu.hio.no"
  }

# Default PHP memory model is too small
  { /etc/php.ini
    ReplaceAll "^memory_limit =.*" With "memory_limit = 16M"
    AppendIfNoSuchLine "memory_limit = 16M"
  }
```

The **processes** rules control the cfengine, MySQL and Apache daemon processes:

```
processes:
  "cfservd" restart /var/cfengine/bin/cfservd
  "cfenvd" restart "/var/cfengine/bin/cfenvd -H -T"
  "mysqld" restart "/etc/init.d/mysql restart"
  "httpd.conf -DSSL -DSVN_VIEWCV"
    restart "/etc/init.d/apache2 startssl"
  "httpd.conf -DSVN_VIEWCV" signal=term
```

The policy file includes some **shellcommands** rules that synchronize the local time with an NTP server, back up the SQL databases, and handle database corruption instances, as detected by the **ReadTCP** function in the **control** section, whose output determines whether the **viewcvs_error** class is defined:

```
shellcommands:
  "/usr/sbin/ntpdate ntp.cfengine.org > /dev/null"
  "/usr/bin/mysqldump > /files/mysql.backup" ifelapsed=480

 viewcvs_error::
  "/etc/init.d/apache2 stop"
  "cd /files/projects; svnadmin recover Cfengine-2"

alerts:
 viewcvs_error::
  "DB corruption error from web server; attempting recovery."
```

An alert is also generated when the Web server probe returns an error.

4.3 A Site Policy File Suite

In this section, we consider a small version of the set of master policy files that a site might deploy on every system that runs cfengine. We have still had to truncate the complete set of rules significantly, for space reasons, but even this subset will give you a sense of what a full-featured, mature cfengine deployment is like.

4.3.1 Main Policy File: cfagent.conf

We'll begin with the *cfagent.conf* file. As usual, we use only hard classes, or classes that might be true of the environment, in this file, since user-defined classes in imported files are never seen in the parent file. Moreover, even though we could, in principle, use **AddInstallables** to make user-defined classes visible, these would get defined too late to be of any use in import decisions.

Note that the configuration file names we've chosen to use here are arbitrary; other names may make more sense in your environment.

```
# cfagent.conf
import:
  any::
    cf.classes
    cf.main

  !matrix:: # matrix is too specialized
    cf.site
    cf.services
    cf.users

  linux.!matrix:: cf.linux
  solaris:: cf.solaris
  hpux:: cf.hpux
  freebsd:: cf.freebsd

#  Subsystem-specific policy files
  any:: # files use classes internally to limit scope
    cf.www
    cf.ftp
    cf.mail

#  Host-specific policies
  labs:: cf.rebuild_labs
  matrix:: cf.matrix
#  end cfagent.conf
```

4.3.2 Class Definitions: cf.classes

Here is the class definition file, *cf.classes*:

```
# cf.classes
classes:
   x86 = ( i386 i486 i586 i686 )  # Intel boxes

   lab1= ( daystrom panzer faust vorlon nirvana nevermore
           delenn voyager borg mulder duke bajor collective
           arrakis axis valis ubik )
   lab2 = ( winter mute roog zhora deckard usul tetsuo zodiac
            ix thistledown jart tleilax axolotl giediprime )
   labs = ( lab1 lab2 )

   WWWServers  = ( nexus orion )
   FTPServers = ( orion mercury acrasia )
   PrimeServers = ( nexus )
   SlaveServers = ( quetzalcoatal cube )
   NameServers = ( PrimeServers SlaveServers )
   securehosts = ( nexus matrix sigmund )
   backupserver = ( dax )

   OnTheHour = ( Min00_05 Min05_10 Min10_15 Min15_20 Min20_25 )
   OnceaDay    = ( Hr00.OnTheHour )
   PeakTime     = ( Hr10 Hr11 Hr12 Hr13 Hr14 Hr15 )
   CheckIntegrity = ( Hr06.OnTheHour )
#  end cf.classes
```

4.3.3 Global Settings and Network Interface Policies: cf.main

Here is the **control** section from *cf.main*, the file that contains settings and rules used by every system under cfengine's control:

```
# cf.main
control:
   Access  = ( root ) # Only root should run this

   domain  = ( iu.hio.no )
   sysadm = ( cfengine@iu.hio.no )
   smtpserver = ( nexus.iu.hio.no )

   timezone = ( MET CET )

   Repository = ( /var/spool/cfengine )
   SpoolDirectories = ( /var/spool/cron/crontabs )
```

```
 labs::
   SplayTime = ( 15 )

 any::
   IfElapsed  = ( 15 )
   ExpireAfter = ( 240 )
   SensibleSize = ( 1000 )
   SensibleCount = ( 5 )
   EditfileSize = ( 40000 )

#  Security settings
   NonAlphaNumFiles = ( on ) # Warn on filenames w/ bogus chars
   SuspiciousNames = ( .mo lrk3 lkr3 nuke rootkit cloak zap
     icepick toneloc .ek wzap clnlog sniff.pid sp.pl )
   ChecksumDatabase = ( /var/cfengine/db/cf.db3 )

   AddInstallable = ( rootfull labupdate dnsupdate syslogdhup )
   cfgmaster = ( pandora )
   cfgsrc = ( /usr/local/sbin/master )

   actionsequence = (
     editfiles
     copy
     checktimezone
     resolve
     netconfig
     shellcommands
     links.Prepare
     files.Prepare
     directories
     links.Rest
     tidy.IfElapsed120.ExpireAfter240
     disable
     editfiles
     files.Rest
     processes
     )

#  Network configuration
 nexus|quetzalcoatal|haddock::
    interfacename = ( hme0 )   # Newer type of machine
 labs::
   netmask = ( 255.255.254.0 )
```

```
  !labs::
    netmask = ( 255.255.255.0 )
```

The **actionsequence** in this file illustrates two features we have not yet encountered:

- The use of rule type-specific expiration times, as in the **tidy** item. For this item only, the default values for **ifelapsed** and **expireafter** are replaced by the values 120 minutes and 240 minutes, respectively.
- The **links** and **files** items appear twice within the **actionsequence**, followed by a class name suffix in each case. The first appearance will invoke the corresponding stanza with the class **Prepare** defined, and the second appearance will invoke the same stanza with the class **Rest** defined (and **Prepare** not defined). Any rules in these stanzas that are not modified with either class will be executed during the first invocation of that action.

Here is the remainder of the *cf.main* policy file, which sets the directory recursion ignore list, configures the broadcast format and the default route, removes some universally unwanted files, and disables some nearly universally undesirable features:

```
ignore:
#  Pseudo-file systems
   /dev
   /proc
   /devices
   /kernel

#  Directories
   /local/lib/gnu/emacs/lock
   /local/tmp
   /local/lib/tex/fonts
   /local/etc
   /usr/tmp/locktelelogic
   /usr/tmp/lockIDE
   .X*
   .Media*

#  Files and patterns
   ls-R
   mysql.sock
   RootMailLog
   lock
   /usr/bin/[
   !*
```

```
broadcast:
    ones

defaultroute: # Set by subnet
  (192_168_74|192_168_75)::
  192.168.74.1

  192_168_89::
    192.168.89.101

resolve: # Specify different orders for balancing
  nexus::
    127.0.0.1
    192.168.89.26   # quetzalcoatal
    192.168.74.16   # cube

  SlaveServers::
    127.0.0.1
    192.168.89.10   # nexus
    192.168.74.16   # cube

 !NameServers.192_168_89::
    192.168.89.10   # nexus
    192.168.89.26   # quetzalcoatal
    192.168.74.16   # cube

  !NameServers.(192_168_74|192_168_75)::
    192.168.74.16   # cube
    192.168.89.10   # nexus
    192.168.89.26   # quetzalcoatal

tidy: # Clear /tmp and old core files (except for lab 2)
    /tmp/   pattern=*  recurse=inf age=1
    /var/tmp pattern=*  recurse=inf age=1
  !lab2::
    /home  pattern=core recurse=inf age=7


disable:
# Disable hosts.equiv except in sysadmin lab
  !lab1::
    /etc/hosts.equiv
#  Remove nologin file if present during nightly run
  OnceaDay.!securehosts::
    /etc/nologin
#  end cf.main
```

4.3.4 Global Probes, Links, Permissions, and SSH Policies: cf.site

The next file we'll consider is *cf.site*, which also contains rules applying widely across the site. The file begins with some class definitions, followed by the **links** and **disable** stanzas:

```
# cf.site
classes: # Define some probe-based classes
    has_ssh = ( ReturnsZero("/usr/bin/test -f /etc/ssh2/ssh2_config") )
    has_inetd = ( FileExists("/etc/inetd.conf") )
    has_xinetd = ( IsDir("/etc/xinetd.d") )

links:
  Prepare:: # Everything depends on the link
    /local -> /nexus/local

  Rest.!labs::
    /etc/rc2.d/S13kdm ->! /local/etc/init.d/S13kdm

disable:
  any::
    /usr/lib/expreserve

# Remove passwd program except for nexus,daneel,sysadmin lab
  !nexus.!daneel.!lab2::
    /usr/bin/passwd repository=none
```

Here is the **files** stanza, divided between the two passes via classes:

```
files:
  CheckIntegrity.Rest::
    /usr/local owner=root,bin,man,daemon,www-data
    group=root,daemon,bin,staff,www-data,adm,other,sys
    action=warnall mode=u-w recurse=inf
    checksum=md5 syslog=true
    exclude=*.log exclude=.bash_history

  has_inetd.Rest::
    /etc/inetd.conf owner=root group=0 mode=644 action=fixall

  Prepare.!labs::
    /.cshrc m=0644 r=0 o=root act=touch
    /var/spool/cron m=755 act=fixall

  Prepare::
    /etc/ssh2/ssh2_config m=644 o=root g=0 act=fixall
    /etc/ssh2/sshd2_config m=644 o=root g=0 act=fixall
```

Here is the remainder of *cf.site*:

```
copy:
  has_xinetd.labs:: # Make sure labs xinetd files are correct
    /local/etc/xinetd.d dest=/etc/xinetd.d
      owner=0 group=0 mode=444 force=on

shellcommands:
  !has_ssh.!labs::
    “/local/bin/SetupSSH”

disks:
    / freespace=10mb define=rootfull

processes:
    “cfenvd”  restart “/usr/local/sbin/cfenvd” useshell=false
    “eggdrop” signal=kill
    “BitchX”  signal=kill
    “enting”  signal=kill
    “bnc”   signal=kill

  PeakTime::
    “rc5des”  signal=kill
    “stst”   signal=kill

  linux::
      SetOptionString “aucx”

# Kill user-processes > 1 day old.
# At Hr23 works around old ps misfeature
  any.Hr23::
    “Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec”
      signal=kill
      include=ftpd
      include=tcsh
      include=bash
      Many more inclusions ...
      exclude=sshd
      exclude=sowille
      ...
# end cf.site
```

4.3.5 Configuring and Managing Daemons: cf.services

The *cfagent.conf* file next imports the policy file *cf.services*, which manages the most important system server processes—daemons—on the various systems at the site.

The version of the file we present here is shortened significantly, to save space. The actual file contains policies for quite a few more services.

```
#  cf.services
control:
    mastersrv = ( janus.cfengine.org )
    src = ( “/usr/local/master” )

copy:
#  Access control for inetd
    /${src}/etc/hosts.deny server=${mastersrv}
    dest=/etc/hosts.deny mode=644

  labs::
    /${src}/etc/hosts.allow.labs server=${mastersrv}
    dest=/etc/hosts.allow mode=644

    securehosts::
    /${src}/etc/hosts.allow.secure server=${mastersrv}
      dest=/etc/hosts.allow mode=644

    PrimeServers::
    /${src}/etc/dfstab server=${mastersrv}
    dest=/etc/dfstab mode=644

editfiles:
  FTPServers::
    { /etc/shells
      EmptyEntireFilePlease
      AppendIfNoSuchLine “/bin/tcsh”
      AppendIfNoSuchLine “/bin/bash”
      AppendIfNoSuchLine “/bin/false”
    }

    {  /etc/syslog.conf # Configure central log host
      AppendIfNoSuchLine “*.warning @loghost”
      DefineClasses “syslogdhup”
    }

processes:
  syslogdhup::
    “syslogd” signal=hup

  NameServers::
    “named” restart “/usr/sbin/named -u dns”
      useshell=false inform=true
  sunos_5_6::
    "identd" restart "/local/sbin/identd"

disable: # Rotate key syslog files
  Sunday.OnceaDay::
    /var/log/messages rotate=6 size=500m
  Day1.OnceaDay::
    /var/log/sulog rotate=4 size=500m
# end cf.services
```

4.3.6 User Account Policies: cf.users

The *cf.users* file defines policies related to user accounts, including checking for potential security problems, removing junk files, and (then) performing nightly backups of user files.

Here is the first part of the file. It defines a long list of patterns for items to be excluded from backup operations, two filters for locating potential security problems, and the **tidy** stanza:

```
# cf.users
control:
# backup exclusions
  excludecopy = ( *.EXE *.avi *.ZIP *.AVI *.MP3
                  *.mp3 *.o *.dvi *.rar ... )
    backupdirs = ( bkupAH:bkupIN:bkupOZ )
    SensibleCount = ( 20 )

filters:
    { history # Shell history = /dev/null
      NameRegex:   ".*history"
      IsSymLinkTo:  "/dev/null"
      Result:    "IsSymLinkTo.NameRegex"
      DefineClasses: "historyalert"
    }

    { setuid # SetUID/SetGID
      Owner:  "root"
      Group:  "0"
      Mode:  "+6000"
      Result: "(Owner|Group).Mode"
    }
```

```
tidy:
  emergency|labs:: # emergency class used for ad-hoc runs
    /home include=.rhosts age=0 inform=on
    /home include=core r=inf age=1
    /home include=a.out r=inf age=1
    /home include=*.o r=inf age=7
    Many more patterns ...
    /home/.netscape/cache include=*
    recurse=inf age=3 type=atime

#  Make sure backup disks don't get full
    backupserver.Hr17.OnTheHour::
        /${backupdirs} include=* recurse=inf age=14
```

The **copy** stanza performs the actual backup operations as well as removing world write access from any files that have these permission bits set.

```
files:
  Rest.Hr01.OnTheHour::
    /home mode=-6002 recurse=inf action=fixplain
      syslog=on filter=setuid
    /home filter=history action=alert
    /home mode=o-w include=* recurse=inf
      action=fixplain syslog=on

copy:
  labs.Hr03.OnTheHour::
    /home/AH dest=/backup/backupAH
    recurse=inf size=<4mb typecheck=false
    backup=false # Don't backup the backup!
    action=silent

  labs.Hr04.OnTheHour::
    /home/IN dest=/backup/backupIN
     Same as previous ...

  labs.Hr05.OnTheHour::
    /home/OZ dest=/backup/backupOZ
     Same as previous ...
# end cf.users
```

4.3.7 OS-Specific Policy Files: cf.freebsd as an Example

The next section of the *cfagent.conf* policy file contains several operating specific files. Here is a simple example designed for FreeBSD systems:

```
# cf.freebsd
control:
   bsdmaster = ( devil )

links: # Restore finger habits
   /usr/spool -> /var/spool
   /usr/local/bin/perl  -> /usr/bin/perl
   /usr/lib/sendmail -> /usr/sbin/sendmail

files:
   /usr/tmp mode=1777 owner=root action=fixall

editfiles:
   { /etc/crontab # Disable standard cleanup scripts
     HashCommentLinesContaining "daily"
     HashCommentLinesContaining "weekly"
     HashCommentLinesContaining "monthly"
   }

directories:
   /var/spool/VirtualLight o=root g=other mode=755

copy:
   /etc/printcap.client server=${bsdmaster}
     dest=/etc/printcap mode=0644

shellcommands:
     "/usr/local/sbin/BSD-pw-update"

  OnceaDay::
     "/usr/libexec/locate.updatedb"
     "/usr/bin/makewhatis $MANPATH"
# end cf.bsd
```

4.3.8 Subsystem-Specific Policy Files: cf.ftp as an Example

The final group of policy files specified in *cfagent.conf* manage specific subsystems/ features and generally apply to special-purpose servers. Our example is the policy file *cf.ftp*, which defines policies for FTP servers. The entire file is processed for hosts in the **FTPServers** class (defined in *cf.classes*) and ignored for all other hosts. The latter is accomplished by defining an alternate **actionsequence** by including the **FTPServers** class throughout the policy file.

Here is the first part of the file, containing its **control** section, as well as the **files** and **directories** stanzas:

```
# cf.ftp
control:
  FTPServers::
    ftp = ( /usr/local/ftp )
    uid = ( 99 ) # ftp user
    gid = ( 99 ) # ftp group

directories:
  FTPServers::
    ${ftp}/pub     mode=644 owner=root group=other
    ${ftp}/etc     mode=111 owner=root group=other
    ${ftp}/dev     mode=555 owner=root group=other
    ${ftp}/usr     mode=555 owner=root group=other
    ${ftp}/usr/lib  mode=555 owner=root group=other

files:
  FTPServers::
    ${ftp}/etc/passwd mode=644 o=root action=fixplain
    ${ftp}/etc/shadow mode=400 o=root action=fixplain
    ${ftp}/pub r=inf mode=644 o=${uid} action=fixall
```

Here is the **copy** stanza, which ensures that all required items are copied from the real file system to the FTP **chroot** directory tree:

```
copy:
  FTPServers::
     /bin/ls dest=${ftp}/usr/bin/ls mode=111
      owner=root type=checksum syslog=true

    /etc/netconfig dest=${ftp}/etc/netconfig mode=444 o=root

    /devices/pseudo/mm@0:zero dest=${ftp}/dev/zero m=666 o=0
    /devices/pseudo/clone@0:tcp dest=${ftp}/dev/tcp m=444 o=0
    /devices/pseudo/clone@0:udp dest=${ftp}/dev/udp m=666 o=0
    /devices/pseudo/tl@0:ticotsord dest=${ftp}/dev/ticotsord m=666 o=0

    /usr/lib dest=${ftp}/usr/lib recurse=2
      mode=444 owner=root backup=false
      include=ld.so*
      include=libc.so*
      include=libdl.so*
      include=libmp.so*
      include=libnsl.so*
      include=libsocket.so*
      include=nss_compat.so*
      include=nss_dns.so*
      include=nss_files.so*
      include=nss_nis.so*
      include=nss_nisplus.so*
      include=nss_xfn.so*
      include=straddr.so*

    /usr/share/lib/zoneinfo
      dest=${ftp}/usr/share/lib/zoneinfo
      mode=444 recurse=2 o=root type=binary
```

The **editfiles** stanza sets policies for the contents of key system configuration files in both the real file system and the FTP directory tree:

```
editfiles:
  FTPServers::
    {  /etc/rc2.d/S72inetsvc
      PrependIfNoSuchLine "umask 022"
    }

    { ${ftp}/etc/passwd
      AutoCreate
      EmptyEntireFilePlease
      AppendIfNoSuchLine
        "ftp:x:${uid}:${gid}:Anonymous FTP:${ftp}:/bin/sync"
    }

    {  ${ftp}/etc/group
      AutoCreate
      EmptyEntireFilePlease
      AppendIfNoSuchLine "ftp::${gid}:"
    }

    {  ${ftp}/etc/shadow
      AutoCreate
      EmptyEntireFilePlease
      AppendIfNoSuchLine "ftp:NP:6445::::::"
    }

    {  /etc/passwd
      AppendIfNoSuchLine
        "ftp:x:${uid}:${gid}:Anonymous FTP:${ftp}:/bin/sync"
    }
```

```
   { /etc/group
     AppendIfNoSuchLine "ftp::${gid}:"
   }
# end of cf.ftp
```

The final item in the *cfagent.conf* file is a policy file used to manage the host *matrix*, which requires substantial individualized attention. We will not examine it here.

4.4 Gathering Data from Many Hosts

Sometimes you want to collect and collate data from a number of different hosts in the network by collecting a known file and placing it on a "repository host" in a special directory. For instance, you might be collecting log files or other monitoring statistics.

The first instinct of many administrators is to use a push solution, giving up the basic cfengine security principle. Cfengine will not allow this, however, and you might start thinking of embedding shell commands to perform this task. This is not necessary. Let us suppose that you have a centralized repository host and you want to collect a file from all of the other clients.

- Run **cfservd** on every client. Its configuration file will need to include an **admit** section that gives the repository host access to the appropriate local files and directories.
- Use a *cfagent.conf* entry on the repository host to collect a file from the list of hosts. We use list iteration to accomplish this.

Policy Example 21: Iterating Over a List of Hosts

```
control:
   actionsequence = ( copy )

   list = ( host1:host2:host3:host4 )
# Another method:
#  list = ( SelectPartitionNeighbours(/dir/cfrun.hosts,#,random,4) )

copy:
  repository_host::
   /var/cfengine/database dest=/depot/$(this)/db server=$(list)
```

Notice that the special variable **$(this)** holds the value of the current server during iterations. Using it will cause the file */var/cfengine/database* on *host1* to be placed in */depot/host1/db* on the repository host, and so on for each host in the list.

5. Extending Cfengine: Modules and Methods

Shell commands are simple and straightforward, but they do not allow any useful communication between **cfagent** and the script being executed. We would like to be able to take advantage of cfengine's classes in scripts, and we would like to be able to return values and classes from scripts to **cfagent**.

5.1 Modules

Modules fulfill the purposes mentioned above. Modules are simply programs, written in any language, that follow a protocol. They are stored in the */var/cfengine/modules* directory. They can do whatever you like, and cfengine will interpret their output (sent to standard out) appropriately. Using modules we can:

- Send parameters and classes.
- Return macro values and classes.

There are two kinds of modules, both of which are added to the control part of a policy:

- Preparatory modules, PrepModules, executed during parsing.
- Action sequence modules, executed after parsing.

PrepModules are executed immediately once parsed. This allows us to read in data that will be used during the parsing of a program, e.g., set variables and classes. Action sequence modules are only run after parsing, at run time, in the relevant part of the **actionsequence**.

Here is an example illustrating how to invoke the two types of modules:

```
control:
   gotinit = ( PrepModule(startup2,"arg1 arg2") )
   actionsequence =
      (
      copy
      "module:mymodule arg1 arg2"
      editfiles
      )
```

The variable **gotinit** stores the exit status of the *startup2* script.

Note that the module name begins with "module:"; this naming convention is required for modules listed in an **actionsequence**.

5.1.1 Modules Protocol

A module can do anything you like, but it should be convergent, for the reasons we have already discussed.

Often we would like to benefit from cfengine's knowledge of the class environment. Cfengine tries to package all of its classes into a shell environment variable called **ALLCLASSES**. However, passing **ALLCLASSES** to scripts could be very long! Most UNIX variants balk at environment variables larger than 2048 bytes, so the list is truncated. The variable *allclasses* is also defined in all parts of cfengine and is guaranteed not to be larger than 2048 bytes.

However, if you have very long class lists, it is better to read the data as a file. The file */var/cfengine/state/allclasses* contains a list of all defined classes, one per line.

Any output from the module is echoed back to **cfagent** and will be quoted as output from the script, except for lines that begin with either +, -, or =, which are processed as variable and class declarations. For example, if the module output looks like this:

```
=var5=what do you know
+wedidit
-problem
more output follows, which is just text
```

then **cfagent** interprets the first three lines as commands to set a variable or class.

The **+** symbol says to define a class, and the **-** symbol means to undefine a class. The **=** symbol says to define the variable string that follows. Thus, in the example above, we set a variable named *var5* to the value **what do you know**. We also define the class *wedidit*, and we remove the class called *problem* (if it exists). Note that deleting a class that is not defined has no effect.

5.2 Methods

Modules are good for tasks such as querying databases, interacting with third-party software, and the like. The disadvantage of using modules is that we are not automatically protected by all of cfengine's safety features. For that reason, cfengine subroutines were introduced. These are called methods.

Methods are:

- A form of sub-program execution.
- Executed as a separate process, sharing the same code text (relatively lightweight processes).

- Encapsulated private variable space (closures).
- Able to receive and return both variables and classes on demand.

A method is a cfengine program, not a shell script. It behaves in every way like a normal cfengine program and has all of the same features and limitations. It can contain/wrap other shell scripts.

5.2.1 Invoking a Method

Methods are invoked in a separate **methods** stanza within a policy file. Here is the general procedure for calling a method:

```
control:
   actionsequence = ( ... methods ... )

methods:
   name( parameter1,parameter2,...)
   action=filename # File containing the method code
   sendclasses=class-list
   returnvars=variable-list
   returnclasses=class-list
```

Methods may also include options for sandboxing (as for **shellcommands**), as well as the **server** option for specifying the computer upon which to run the method (allowing for remote execution). See the *Cfengine Reference* for details on these options.

Here is a concrete example of invoking a method:

```
control:
   actionsequence = ( methods )

classes:  # A defined class is required by alerts
   every = ( any )

methods:
   MethodTest("We sent some text","/etc/motd")
   action=cf.methodtest
   returnvars=retval
   server=localhost

alerts:   # Display message showing the return value
  every::
   "Method returned $(MethodTest.retval)"
```

The actual method code is kept in the file called *cf.methodtest*, which must be placed in the */var/cfengine/modules* directory. Let us look at a simple example showing how data is sent back and forth.

In this example, we send the method two character-string arguments. We end by defining an alert that prints the value returned by the method call.[19] The **returnvars** option tells **cfagent** that it should name the return value *retval*, which is placed in the nonconflicting namespace *MethodTest* to signify that it comes from that method.

If we run **cfagent** with this policy file, we get the following output:

```
cfengine:: Method returned We sent some text ... and got some back.
```

In order to understand this output, we now turn to the method configuration itself.

5.2.2 Method Declaration

The file that contains the code for the method differs only slightly from an ordinary cfengine input file. It contains the additional **control** directives **MethodName** and **MethodParameters**, which indicate that the current file is to expect a bundle of information from a parent and that it will return a bundle of information to its parent when it completes.

Here is the configuration file for our *MethodTest* method:

```
control:
    MethodName = ( MethodTest )
    MethodParameters = ( value1 value2 )
    var1 = ( "${value1} ... and got some back." )
    actionsequence  = ( editfiles )

classes:  # Class required by alerts
  every = ( any )

editfiles:
    { ${value2}
      AppendIfNoSuchLine
        "Enjoy your time in Oslo, the Official Cfengine Test City."
    }

alerts:
  every::
    ReturnVariables("${var1}")
```

The **MethodName** declaration confirms the name of the method to the parent. The **MethodParameters** directive tells the method how it should interpret the arguments that were transmitted to it. The arguments are placed in new variables called *value1* and *value2*.

[19] Alerts are discussed in detail in Chapter 6.

The method code calls only **editfiles**, which appends some text to the file specified as the method's second argument (if it is not already present). This is the only real work performed by this sample method. Finally, an alert is generated which sends a variable back to the parent. In this case, the value is that of the local variable *var1*, which consists of the method's first argument concatenated to " ... and got some back.", resulting in the displayed message we saw previously.

5.3 Example: Generating the Password and Shadow File

In this section, we considered an extended, real-world example using the cfengine methods and modules capabilities.

There are many ways to arrange for password distribution. Nearly all of them are fraught with some kind of trouble. Here is a simple approach to selecting a number of users from another UNIX host's password file.

This example uses a module and a method to solve a subset of the password distribution problem. It uses the following files in */var/cfengine*:

- *inputs/cfagent.conf*: Main policy file
- *modules/cf.passwd*: Method definition file
- *modules/module:getusers*: Module shell script

5.3.1 The Policy File

Here is the key part of the policy that handles password and shadow file generation. It calls the method named *ImportPasswords,* which requires no arguments.

```
control:
   actionsequence =
     ( shellcommands processes methods files )

shellcommands:
   "/usr/sbin/ntpdate $(ntphost) > /dev/null"

processes:
   "portmap" restart "/sbin/portmap"
   "rpc.mountd" restart "/usr/sbin/rpc.mountd"
   "rpc.nfsd" restart "/usr/sbin/rpc.nfsd"
   "cfservd" restart "/var/cfengine/bin/cfservd"
   "mysqld" restart "/etc/init.d/mysql restart"

methods:
   ImportPasswords(void)
   action=cf.passwds
   server=localhost
```

```
files:
   /etc/passwd owner=root group=0 mode=0644 action=fixplain
   /etc/shadow owner=0 group=0 mode=600 action=fixplain
#  Make it hard for fingers to change the passwords
   /usr/bin/passwd mode=0400 action=fixplain
```

This policy file ensures that the local time is synchronized with the NTP master (so that update decisions are made based on a correct time value) and that required daemons are running. It then invokes the *ImportPassword* method. Finally, the policy sets the ownership and mode for the password and shadow files and turns off the execute bits for the **passwd** command to prevent changes.

5.3.2 The ImportPasswords Method

The password file entry importing method uses a module to collect a list of authorized users from a database. It then collects complete password and shadow files from a trusted source, deletes all lines that do not begin with the name of an authorized user, and merges the remaining lines into the existing password file, replacing any entries that are already there. Finally, it corrects the users' home directory entries and deletes its work files.

```
control:
   MethodName = ( ImportPasswords )
   MethodParameters = ( none )
   actionsequence =
     ( copy module:getusers editfiles directories tidy )
   Split = ( "," )
   editfilesize = ( 0 ) # unlimited

   # Locations of remote source and local work files
   srcserver = ( pmaster.cfengine.org )
   srcpasswd = ( /master/etc/passwd )
   srcshadow = ( /master/etc/shadow )
    tmppwd = ( /var/run/workfile1 )
   tmpshad = ( /var/run/workfile2 )

#    File generated by module:getusers
   ufile = ( /var/run/userlist )
   ulist = ( ReadList("${ufile}","lines","#","1000") )

copy:      # Copy remote master copies
   ${srcpasswd} server=${pmaster}
     dest=${tmppwd} mode=600 type=checksum

   ${srcshadow} server=${pmaster}
     dest=${tmpshad} mode=600 type=checksum
```

```
# module:getusers runs and creates /var/run/userlist

editfiles:
#    Remove all entries not in the generated user list
   {  ${tmppwd}
       DeleteLinesNotStartingFileItems “$(ufile)”
   }

   {  ${tempshad}
       DeleteLinesNotStartingFileItems “$(ufile)”
   }

#    Generate the real files from the work files
   {  /etc/passwd
       DeleteLinesStartingFileItems “${ufile}”
       AppendIfNoSuchLinesFromFile “${tmppwd}”
   }

   {  /etc/shadow
       DeleteLinesStartingFileItems “${ufile}”
       AppendIfNoSuchLinesFromFile “$(tmpshad}”
   }

directories:  # Configure user home directories
   /home/${ulist} owner=LastNode                    LastNode = final path component.

tidy:
   /var/run include=workfile* include=userlist* age=0
```

For space reasons, we have not been as careful in checking the results of each operation before proceeding with the next one in this policy file.

5.3.3 The module:getusers Shell Script

This script simply generates a list of usernames of the users who are to be given accounts on the custom machines. We assume that this list is a subset of the users on the regular, noncustomized hosts.

```
#!/bin/sh

list=/var/run/userlist

# Extract names from a database
mysql -BN -h sqlserverhost -u nobody -D users \verb+\+
  -e “select account,mygroup from mytable \verb+\+
  where mygroup=’chosenfew’ and status=’ENABLED’” \verb+\+
  | cut -f1 > $list
```

```
# Append some additional names
echo mark >> $list
echo aeleen >> $list
```

The script file will need to have its execution bits set in order for it to be successfully executed.

6. Host Monitoring and Anomaly Detection

Traditionally, administrators have viewed management software as being about monitoring and presenting data about systems to humans. This has been part of a long-running philosophy of servicing components: wait until something fails and then replace it, please, Dave. Cfengine is not really about monitoring, but it certainly *is* about the opposite of getting involved with humans. Moreover, there is ongoing research focused on using cfengine to simplify the problem of system monitoring.

First of all, let us clear up a misconception: cfengine does not care specifically about anomaly detection in the sense of *intrusion detection.* This is because there is no provable correlation between anomalies and intrusions—the link between these has been oversold by security people. Cfengine *is* interested in the detection of resource anomalies, regardless of their cause. It can autonomically monitor the usage of a subset of system resources: disk, CPU, number of users, network services, etc. Why would a tool like cfengine bother to do this?

There are obviously advantages to monitoring such resource usage. We can better tune and configure a system if we know how it works on a good day and compare this to how it fails on a bad day. On the other hand, in general, if you don't know what you are looking for in a system, there is little point in collecting reams of data about it, via **cfenvd**'s potentially highly resource-intensive process.

The answers to the existential questions are by no means clear, and they are still the subject of much research. It is simply not known whether there is much point in having automated systems try to monitor and regulate systems at the level of configuration management. Certainly there are examples where low-level regulation can improve the efficiency of services, e.g., by deploying more resources in response to a heavy load, but whether there is much we can do about automating responses in configuration is unclear. At the time of this writing, we are exploring the use of this kind of feedback for load balancing and overload (denial-of-service) protection, but the full story will have to be told elsewhere.

6.1 Autonomic Computing

Whatever we lack in understanding of the issues, we must make up for by trial and experience in the field. One of the research aims of cfengine is to move toward a vision of autonomic computing, as a proof (or disproof) of concept. The idea of self-regulating systems has gone through several conceptual shifts over the years, including the idea of artificial immune systems. The basic idea is that we should borrow ideas from biology to allow computers to detect and fix their own maladies. Based in this analogy alone, we think that it will be helpful to detect anomalous behavior and respond to it.

Several well-known systems already exist for recording the immediate levels of system data from the Simple Network Management Protocol (SNMP). Cfengine will also shortly have SNMP capabilities, but not in the manner of the well known Multi Router Traffic Grapher (MRTG) program, as cfengine's philosophy is to reduce the amount of information being thrown at humans, not compound it.

At the present time, cfengine can detect certain kinds of anomalies and respond to them, report them, and so on.

6.2 Alerts: Some Basics About Warnings

The simplest response to an anomaly is to issue an alert. The **alerts** section of cfagent's configuration is used for this. **alerts** simply prints messages based on class membership, and every entry within this stanza must be preceded by a explicit class specificative. The class **any** is always defined, so you are not allowed to place an alert in this class, because it would always give rise to a message. Since this is probably an expensive mistake if you have 10,000 hosts, **cfagent** makes it a little difficult for you.

You can test alerts using some simple rules like these:

```
classes:
   jambalaya = ( any ) # a class that is always true

alerts:
  jambalaya::
    "Gumbo!"
```

In general, alerts are most useful if they are issued only in unusual circumstances. For example, let's see what happens when a variable crosses a threshold:

```
# Interface variables from cfenvd
classes:
   lt = ( LessThan(${average_users},6) )
   gt = ( GreaterThan(${average_otherprocs},60) )
```

```
alerts:
  lt:: "$(a) LESS THAN $(b)"   # shut down for the night?
  gt:: "$(a) GREATER THAN $(b)"  # security incident?
```

The special variables in the messages for these alerts correspond to the function call arguments used to define the corresponding class. Thus, *$(a)* in the first alert will print the *average_users* value specified to **LessThan,** and *$(b)* in the second alert will print the second argument to **GreaterThan**, “60”.

Alerts can also be channeled directly to **syslog**, to avoid extraneous console messages or email:

```
alerts:
  lt::
    SysLog(LOG_ERR,"Test syslog message")
```

We shall use alerts more in this chapter.

6.3 The cfenvd Daemon

The basis of resource monitoring is the **cfenvd** daemon. **cfenvd** requires no configuration. It makes use of tools existing on your system for monitoring—commands such as **ps** and **netstat**. If you have **tcpdump** installed, **cfenvd** can use it to monitor traffic (**cfenvd -T**). Presently, it will also be able to interface with the **scli** SNMP interface software. It updates its measurements every two minutes, a carefully measured balance between too often (heavy on resources) and too seldom (missing important information).

Using a smart learning algorithm, **cfenvd**:

- Learns the behavioral trends in each computer over weeks.
- Evaluates the current state of the resources against learned averages.
- Classifies the current state against learned averages into a basic ontology of performance.

When **cfenvd** is active, it records data in */var/cfengine/state*. Later, when **cfagent** intermittently starts, it reads classes and variables about the current state of resources from a file *env_data* in the same directory. These are of the form:

$(value_tcp_in)	*Most recent measured value*
$(average_www_out)	*Weekly mean value*
$(stddev_userprocs)	*Weekly standard deviation*
rootprocs_high_anomaly	*Class related to the number of root processes*
loadavg_high_dev2	*Class related to the load average*

The first two items show the use of the **_in** and **_out** suffixes with metrics that distinguish incoming and outgoing connections.[20] The latter two items illustrate the automatic classes that are defined based on comparing current resource usage with normal values. The second and third components of these classes, which function syntactically as suffixes to the metric keyword, have the following meanings:

low, normal, high	*Current value is <, ≈ or > normal.*
dev1, dev2, anomaly	*How far current value deviates from norm: 1, 2 or 3 standard deviations.*

Like everything in cfengine, alerts about anomalies are subject to policy decisions. Here is a sample **cfagent** configuration for alerting about unusual activity:

```
alerts:
# The class anomaly_hosts matches hosts collecting anomaly data
    anomaly_hosts.UserProcs_high_dev2::
    "UserProc anomaly high 2 dev on $(host)/$(env_time) \
    current value $(value_otherprocs) av $(average_otherprocs) \
    pm $(stddev_otherprocs)"
    ShowState(procs)

  entropy_www_in_high.anomaly_hosts.www_in_high_anomaly::
    "HIGH ENTROPY Incoming www anomaly high anomaly dev!!\
     on $(host)/$(env_time) current value $(value_www_in) \
     av $(average_www_in) pm $(stddev_www_in)"
    ShowState(incoming.www)

  anomaly_hosts.smtp_out_high_dev2::
    "Outgoing smtp anomaly high 2 dev on $(host)/$(env_time) \
     current value $(value_smtp_out) av $(average_smtp_out) \
     pm $(stddev_smtp_out)"
    ShowState(outgoing.smtp)
```

[20] The **ShowState** function, illustrated and described below, uses the initial components **incoming** and **outgoing** to distinguish connection types, e.g., **incoming.www**.

Table 6.1 lists the available metrics tracked by **cfenvd**.

Metric	Description	Port	_in and _out accepted?
cfengine	Cfengine-related traffic	5308	yes
diskfree	Amount of free disk space		no
dns	Domain name server related traffic	53	yes
ftp	File transfer protocol traffic	21	yes
icmp	Total ICMP traffic (e.g., **ping**)		yes
irc	Internet relay chat protocol traffic	194	yes
loadavg	Current load average		no
netbiosdgm	NetBIOS datagram service	138	yes
netbiosns	NetBIOS name service	137	yes
netbiosssn	NetBIOS session service traffic	139	yes
nfsd	NFS daemon traffic	2049	yes
otherprocs	Number of non-root process		no
rootprocs	Number of processes owned by root		no
smtp	Simple mail transfer protocol traffic	25	yes
ssh	Secure shell remote login protocol	22	yes
tcp	Total TCP traffic	*all*	yes
tcpack	TCP packets with ACK flag set	*all*	yes
tcpfin	TCP packets with FIN flag set	*all*	yes
tcpmisc	All other TCP packets	*all*	yes
tcpsyn	TCP packets with SNY flag set	*all*	yes
udp	Total UDP traffic	*all*	yes
users	Number of logged in users		no
www	World wide web traffic (HTTP)	80	yes
wwws	HTTP protocol over TLS/SSL (HTTPS)	443	yes

Table 6.1 Anomaly Detection-Related Classes

6.4 The ShowState Function

The **ShowState** function reveals more details about the current sample of data that caused the alert we have asked for. For instance, if we ask for:

```
ShowState(incoming.www)
```

in the case of a low-entropy (sharp) anomaly, we might get a report like the following:

```
cf:cube: LOW ENTROPY Incoming www anomaly high anomaly dev!!
on cube/Fri Feb 20 19:57:23 2004
current value 53 av 9.9 pm 16.1
cf:box: ————————————————————————————————————
cf:box: In the last 40 minutes, the peak state was:
cf:box: ( 1) tcp 0 0 128.39.74.16:80 157.158.24.40:4049 TIME_WAIT
cf:box: ( 2) tcp 0 0 128.39.74.16:80 157.158.24.40:3796 TIME_WAIT
cf:box: ( 3) tcp 0 0 128.39.74.16:80 157.158.24.40:3544 TIME_WAIT
cf:box: ( 4) tcp 0 0 128.39.74.16:80 157.158.24.40:4063 TIME_WAIT
cf:box: ( 5) tcp 0 0 128.39.74.16:80 157.158.24.40:4035 TIME_WAIT
cf:box: ( 6) tcp 0 0 128.39.74.16:80 157.158.24.40:3782 TIME_WAIT
cf:box: ( 7) tcp 0 0 128.39.74.16:80 157.158.24.40:3530 TIME_WAIT
cf:box: ( 8) tcp 0 0 128.39.74.16:80 157.158.24.40:3824 TIME_WAIT
...
DNS key: 157.158.24.40 = arm.iele.polsl.gliwice.pl (47/53)
DNS key: 80.203.17.11 = 11.80-203-17.nextgentel.com (1/53)
DNS key: 66.196.72.28 = j3118.inktomisearch.com (1/53)
DNS key: 80.202.77.107 = 107.80-202-77.nextgentel.com (2/53)
DNS key: 80.213.238.106 = ti100710a080-3690.bb.online.no (2/53)

Frequency: 157.158.24.40 |************************************ Frequency:
80.203.17.11  |*
Frequency: 66.196.72.28  |*
Frequency: 80.202.77.107 |**
Frequency: 80.213.238.106 |**

Scaled entropy of addresses = 12.7 %
(Entropy = 0 for single source, 100 for flatly distributed source)
cf:box: ————————————————————————————————————--
cf:box: State of incoming.www peaked at Fri Feb 20 19:57:23 2004
```

Note that the entropy classes generated by **cfenvd** refer to how sharply peaked the distribution of IP addresses or processes is. See how the frequency plot in the example above has a sharply peaked distribution: this is low entropy. A completely flat distribution, where an equal amount of traffic came from every address, would be a maximal entropy distribution.

6.5 The Cfenvgraph Utility

The data cfengine measures is normally hidden from view in a database. It can be instructive to view this information. At present there are only primitive tools available for this. The **cfenvgraph -s** command generates a directory of files that show a weekly snapshot of the system. For extra high resolution, add **-r** to this.

By default, the plots all show the entire week starting from Monday morning and finishing on Sunday evening.

The data files have names in the following forms :

- variable.*q*—This is the latest raw value *q* measured. The format is *x y*.
- variable.*E-sigma*—This is the computed average value with standard deviation. The format is *x y dy*.
- variable.*distr*—This is the distribution about the mean, a frequency histogram, with format *x y*.

These can be viewed, for instance, in **gnuplot**:

```
$ gnuplot
plot “loadavg.q” with lines
plot “loadavg.E-sigma” with errorbars
plot “loadavg.distr” with lines
```

The result is a picture similar to the one in Fig. 6.1.

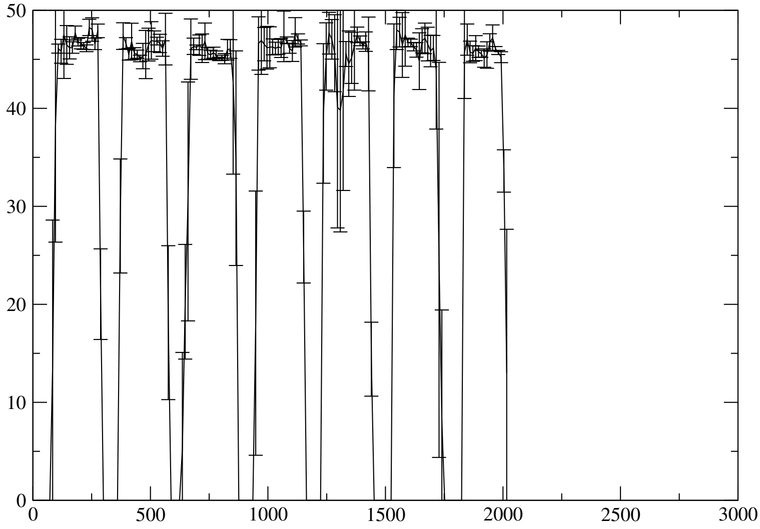

Figure 6.1: Output of cfenvgraph, Viewed with gnuplot

Similar commands can be used with the more advanced **xmgrace** software (which is widely available on the Internet):

```
$ xmgrace -nxy cfenv-average
$ xmgrace -settype xydy www_in.E-sigma -hardcopy -hdevice EPS
$ xmgrace -settype xydy netbiosssn_in.E-sigma -hardcopy -hdevice EPS
$ xmgrace scan\:_mountpoints_disk1.db.cfenv -hardcopy -hdevice EPS
```

Cfenvgraph produces two kinds of graphs: time series and distributions. The time series are graphs of measured value versus time. Time is measured in intervals of 5 minutes so that the scale runs from zero (Monday morning 0:00 hours) to 12 x 24 x 7 = 2016 (Sunday night at 24:00). Each measurement has its own natural scale. Load averages are multiplied by 100 so that the traces do not disappear on the multi-trace graphs.

Distributions measure the scatter about the mean averaged over the whole week. In other words, they are de-trended scatter distributions, placed on a scale from 0 to 100, where 50 is chosen to be the expectation value for the entire week. The vertical scale is a frequency which is not normalized, and hence will be different in every case.

6.6 FriendStatus Alerts

A neighborhood watch scheme is a useful way of monitoring hosts without using a ping or a heartbeart monitor. Cfengine can use its secure communication infrastructure to get hosts to monitor one another in a decentralized way.

A cfengine friend is another host with which cfagent collaborates. Any host that grants us access to files is a friend. Any host to whom we grant access to files is a friend.

Cfengine and **cfservd** record all successful connections between cfengine friends. The **FriendStatus***(n)* function shows known hosts that we have not seen for *n* hours. **FriendStatus(0)** shows hosts that we have not seen for the learned average interval. **LastSeenExpireAfter** determines when friends are judged to have died forever.

From version 2.1.7 on, cfengine includes automatic monitoring of friends. This provides us with a simple way of learning from the network when hosts are down or not responding. Each time cfengine makes contact with a peer (either by opening a channel to **cfservd** on a remote host or by receiving such a connection from a peer agent), it records the time at which it last observed the peer and stores it in a database called “last seen”.

Each time a new connection arrives, cfengine updates an average interval of time since the last time it saw the peer. These can be seen *in verbose mode* (i.e., **cfagent -v**) by using an **alerts** function:

```
alerts:
  myclass::
    FriendStatus(3)
```

In verbose mode, this prints a summary of the last-seen times. It produces alerts in non-verbose mode if any hosts have not been seen for more than three hours.

The output in verbose mode is of the form (lines wrapped for space reasons):

```
cf:myhost: Host duke.iu.hio.no i.e. 192.1.1.65
last hailed us @ [Thu Feb 15 21:01]
cf:myhost: i.e. (3.03) hrs ago, Av 0.10 +/- 0.15 hrs

cf:myhost: Host vorlon.iu.hio.no i.e. 192.1.1.23
last hailed us @ [Thu Feb 15 21:01]
cf:myhost: i.e. (3.02) hrs ago, Av 0.10 +/- 0.15 hrs
```

Each host presents its own perspective on the landscape of friends it normally works with, from the viewpoint of its own policy:

- The first entry tells us that the host 10.39.89.10 answered a request from us only 0.11 hours ago. The average time between our requests to this host is 0.15 hours.
- The second entry says that the host 10.39.89.26 attempted to connect to our **cfservd** process at the stated time, 0.11 hours ago. The expected time between connections from this host was 0.04 hours, so we are overdue.

The **FriendStatus** alert tells us which hosts in our recent memory of connections are overdue according to the time specified in the argument or according to their regular pattern of behavior. If the time specified in the argument of the function is zero, warnings are issued if the time is greater than the expected time. Hosts that have not been seen for more than a week are purged from the database.

The **FriendStatus** alert has a limited value unless we overlay some predictable pattern of interactions between peers, because it is easily confused by anomalies or irregularities. Another way of monitoring hosts is to use a graph theoretical trick that allows every host in the network to watch out for its neighbors.

The following method can be plugged into any configuration. The method is called by adding the following stanza to your configuration:

Policy Example 22: Peer Watch (Part 1)

```
control:
  AddInstallable = ( PeerCheck_done )

methods:
 any::
  PeerCheck(null)
 action=cf.peercheck
server=localhost
returnclasses=done
returnvars=null
ifelapsed=5
```

This policy causes **cfagent** to attempt to execute a method whenever possible, but not more often than every five minutes. The method simply tries to download a small, unimportant file from the remote peers.

The module itself should be placed in the *modules* subdirectory of the main cfengine directory, */var/cfengine/modules/cf.peercheck*. It is listed in Policy Example 23, below.

The idea here is to force hosts to perform a TCP ping of one another to the cfengine port. Each host in its peer group watches over its neighbors. This provides a redundant clique of cross-checkers. Another, less intrusive, approach would be to pick a random host in the clique (the peer leader) to check less frequently. Over time, this amounts to the same thing, but with less resolution.

The method takes a list of hosts (e.g. the one conveniently located in the *cfrun.hosts* file) and partitions it into neighborhood watch areas or *peer groups* of size 4. Each host in its peer group can either select a leader that we will try to contact to see if it is alive, or each host can check all its peers in the neighborhood. Both options are coded above, but the chosen method is to check all neighbors. The method works by simply trying to copy a small file from the hosts. If the copy succeeds, the host is alive and working; if it fails, there is a problem and an error will be signaled.

In this way, hosts in a peer group check each other's integrity.

Policy Example 23: Peer Watch (Part 2)

```
#  A peer to peer ping, using cfengine
control:
   MethodName     = ( PeerCheck )
   MethodParameters = ( null )

   smallfile  = ( /etc/inittab )
   pf      = ( /var/cfengine/inputs/cfrun.hosts )
   peer_leader = ( SelectPartitionLeader(${pf},#,random,4) )
   peers     = ( SelectPartitionNeighbours(${pf},#,random,4) )

   actionsequence = ( copy )
   domain      = ( domain.tld )
   AddInstallable = ( nocontact nocontact_leader)
```

```
classes:
    moduleresult      = ( any )
    check_all_neighbours = ( any )
    check_leader      = ( none )

copy:
  check_leader::
    $(smallfile) server=$(peer_leader)
      dest=/var/cfengine/state/peer_check_$(peer_leader)
      trustkey=true failover=nocontact_leader

  check_all_neighbours::
    $(smallfile) server=$(peers)
      dest=/var/cfengine/state/peer_check_$(peer_leader)
      trustkey=true failover=nocontact

alerts:
  nocontact_leader::
    "Unable to hail host $(peer_leader)" ifelapsed=240

  nocontact::
    "Unable to hail one of hosts $(peers)" ifelapsed=240

  moduleresult::
    ReturnVariables(null)
    ReturnClasses(moduleresult)
```

6.7 File System Scans

Cfengine's time-series analyses can also be applied to the disk file systems on a computing device to learn about the patterns of usage and behavior with regard to storage. When are the peak times during the week for storing data?

Scanning file systems is not something to be done on a continuous basis—it would be too expensive—so this is not done by **cfenvd**. You can ask **cfagent** to perform a file system scan, however, as a matter of occasional policy. The results can be viewed using **cfenvgraph**, just as with other analyses.

Policy Example 24: File System Behavior Scan

```
disks:
  /mountpoints/disk1 scanarrivals=true ifelapsed=1200
```

This would cause **cfagent** to perform a complete recursive scan of the named file system every 1200 minutes, capturing file modification data for later analysis by **cfenvgraph**.

6.8 Interpreting Anomaly Results

Collecting data is all well and good, but we also need to interpret what we find. The simple fact is that no one really knows how to fully interpret data from time-series. This is still an area of active research, in which cfengine's monitoring technology is among the world leaders. Ultimately, the goal is to allow automatic interpretation of data collected.

Let's look at some sample graphs generated from the stored data. For each variable, **cfenvd** stores three sets of data, as explained in section 6.5.

6.8.1 Overview

The graph in Figure 6.2 was generated using **xmgrace**:

```
$ xmgrace -nxy cfenv-average
```

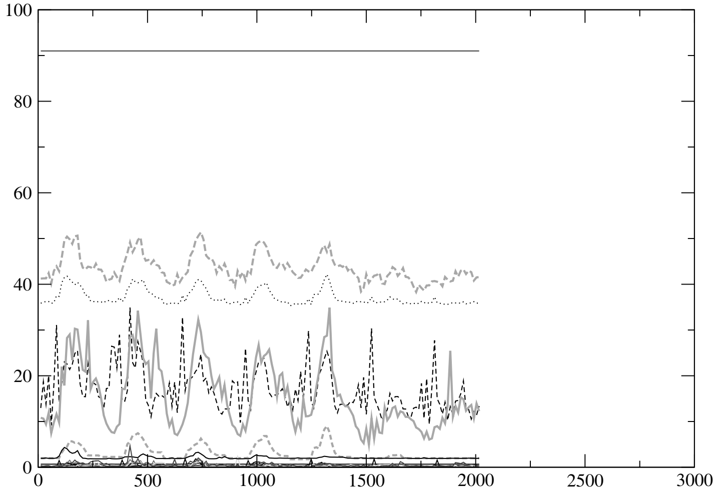

Figure 6.2: Output of cfenv-average, viewed with xmgr

This shows an overview, like a brain-wave trace (EEG) or electrocardiogram (ECG). The advantage of this view is that one can compare the normal behaviors of different resources on a common scale. The disadvantage is that the common scale makes smaller variations too small to see clearly.

6.8.2 Separate Traces

The overview cannot show the details of the traces or easily identify which trace is which. Moreover it shows only the average values, with no idea of how much uncertainty there is in the data.

The graph in Figure 6.3 was generating using **xmgrace**:

```
$ xmgrace -settype xydy netbiosssn_in.E-sigma
```

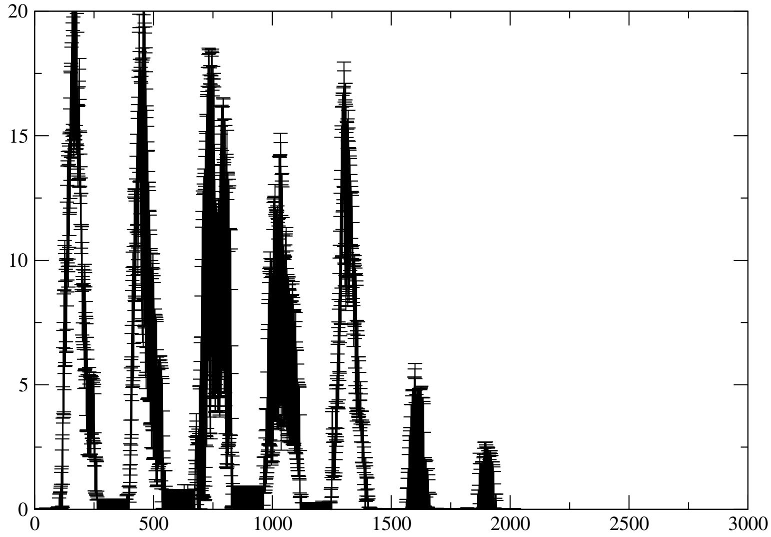

Figure 6.3: Output of netbiosssn_in, Viewed with xmgrace

The time scale on the horizontal axis runs from Monday morning to Sunday evening. The visible peaks are usually centered around the middle of each day. The solid line shows the average value, and the vertical error bars show σ one standard deviation as estimated from the samples. Here we see that the variations of the average are larger than the average size of the error bars (the error bars look small). This means we can be confident that the pattern of resource usage is real.

The second example, in Figure 6.4, was generated using the following command:

```
$ xmgrace -settype xydy www_in.E-sigma
```

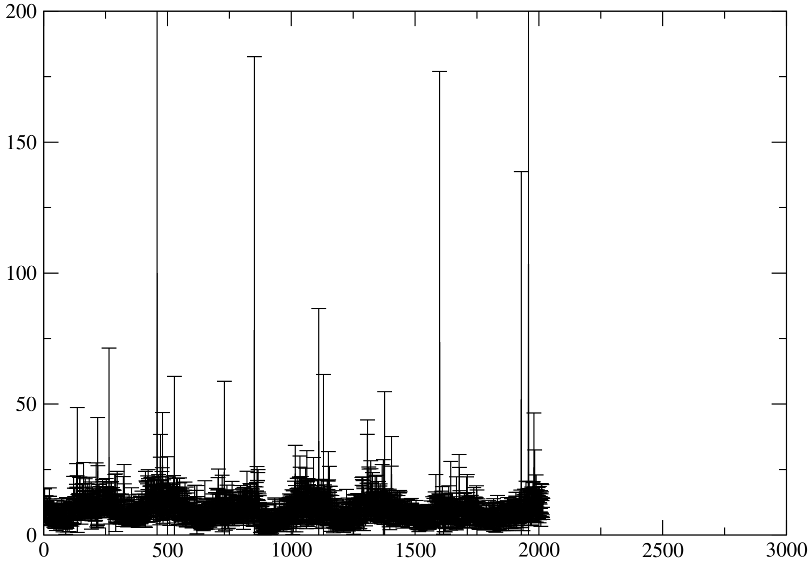

Figure 6.4: Output of www_in, Viewed with xmgrace

In this case, the error bars are larger than the variation in the average, and we have to conclude that this variable is dominated by uncertainty, that is, we cannot make much in the way of predictions about its regular behavior.

6.8.3 File System Scans

The next example, in Figure 6.5, was generated using the following command from the data collected via Policy Example 24:

```
$ xmgrace scan:_mountpoints_disk1.db.cfenv
```

It shows the frequency of changes to the scanned file system at different times during the working week. Once again, we see from this host the familiar pattern of variation, with activity peaking around the middle of each day of the week. This need not be the case, of course. If most of the disk activity were performed as a result of globalized processes at any time of the day or night, we would not see this pattern. What this graph allows us to see is how the pattern of disk activity mirrors other patterns of change, such as load average or the main services, that can be monitored more directly.

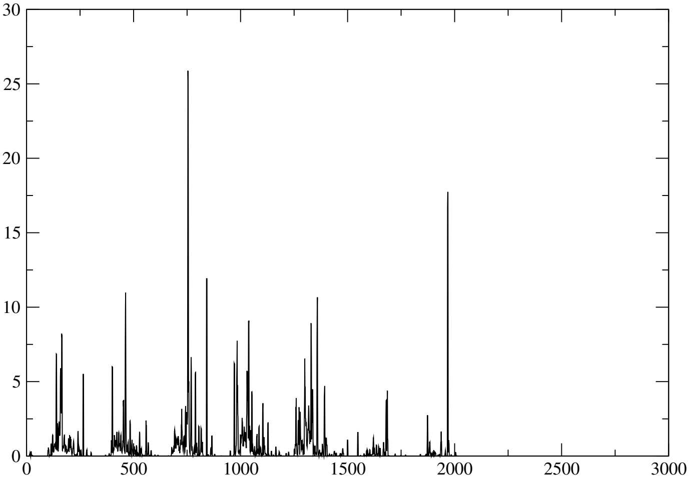

Figure 6.5: Output of scan:_mountpoint_disk1, Viewed with xmgrace

6.8.4 Distributions

The distribution of normal activity levels about the mean also has a significance to host performance. We will illustrate this concept using incoming Web traffic.

The first graph, Figure. 6.6, has a lower bound and tails off at higher values. This means that most of the time the system is underutilized, but there are some bursts of activity, giving a lower number of higher levels.

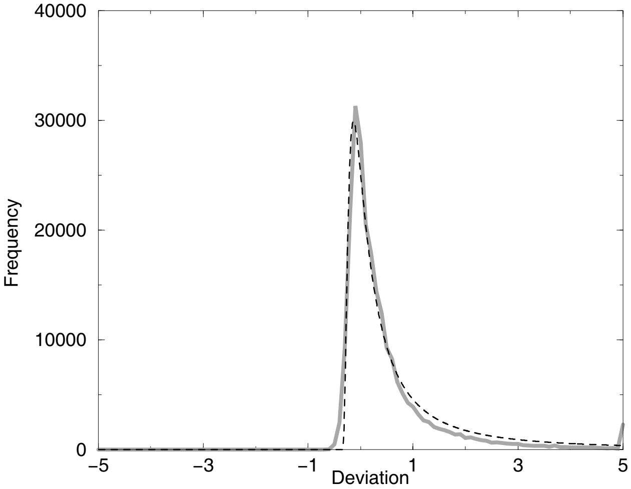

Figure 6.6: Incoming Web Traffic Has a Lower Bound

This curve is smooth and unambiguous. A distribution curve might not always be this perfect, however. Figure 6.7 is an example of a distribution that is more jagged. Real systems often give jagged distributions. These even out only after many, many weeks of data.

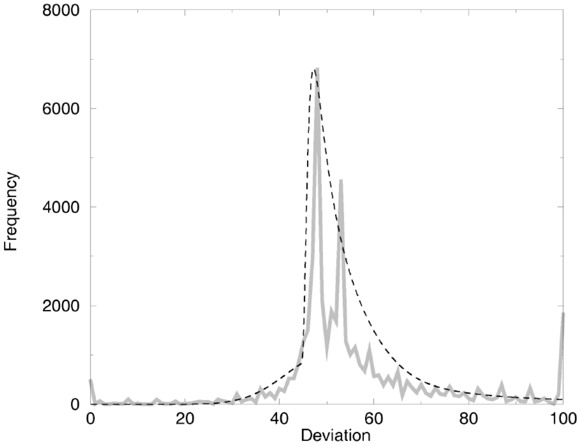

Figure 6.7: Incoming Web Traffic Has a Lower Bound

Next, a two-pronged molar (Figure. 6.8) shows the free disk space on a file system. It has a fascinatingly bimodal distribution, meaning that disk space is generally mostly at one of these two levels. Perhaps the system is cleaned periodically and remains there for some time and then fills up quickly again. How would you explain this distribution?

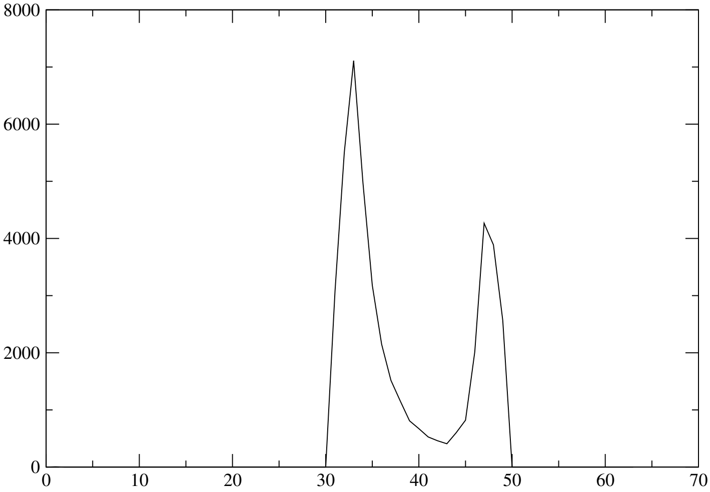

Figure 6.8: Free Disk Space

In Figure. 6.9, an incoming FTP service distribution shows an upper bound. This could be for one of two reasons:

- ❖ The source of the load has an upper bound on its demand.
- ❖ The service has an upper bound on its delivery.

In the latter case, we might want to improve the service capacity.

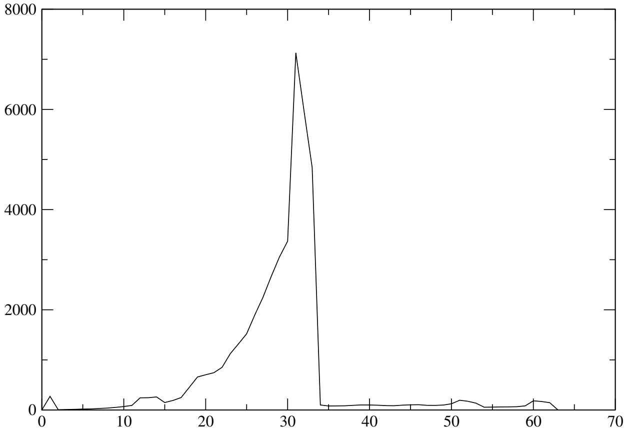

Figure 6.9: Incoming FTP with an Upper Bound

The next distribution is from load average (Figure. 6.10). This shows a fairly healthy spread of values, with no clear tendency toward an upper bound, so no bottlenecks in process performance are apparent.

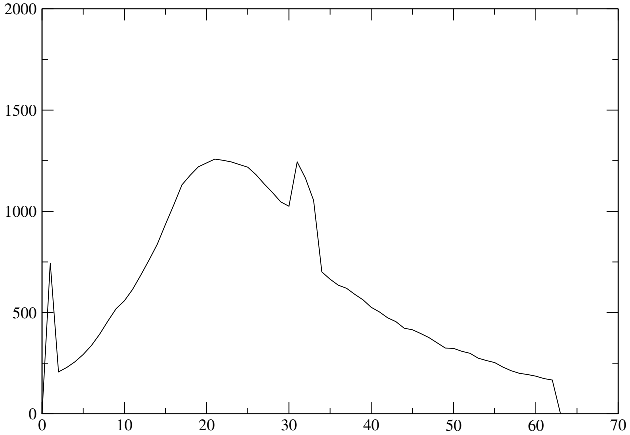

Figure 6.10: Load Average Behavior

In Figure 6.11, a graph of NFS traffic does show an upper limit, however. In this case, it is unlikely to be because of a server bottleneck, but, rather, a result of the fact that there are a fixed number of clients that normally produce the same amount of demand. Sometimes, however, the demand from some is less, perhaps as a result of certain hosts being down or inactive.

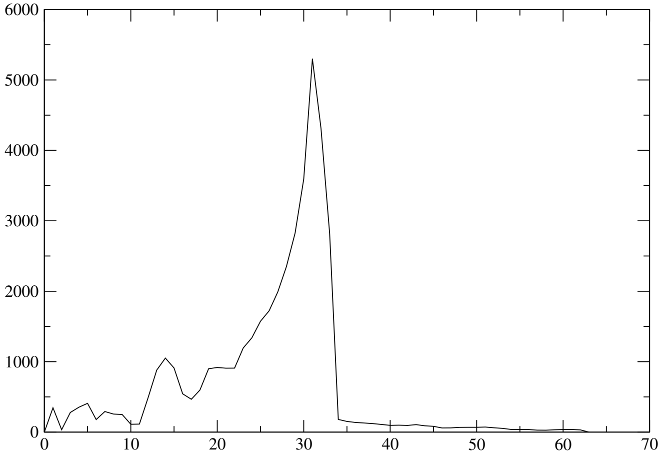

Figure 6.11: Incoming NFS Traffic with an Upper Bound

The final distribution (Figure 6.12), incoming SSH traffic, shows a fairly symmetrical distribution. The large spike in the middle means that the distribution is sharply peaked, but the relatively low number of events makes the distribution jagged.

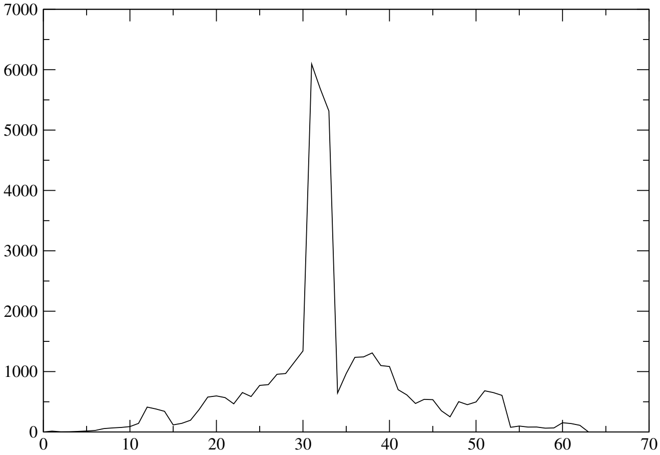

Figure 6.12: Incoming SSH Traffic

The conclusions to be drawn from this data require a certain amount of human interpretation, based on a wider knowledge of the system activity.

In our opinion, monitoring has a limited value. It should either confirm your current understanding of a system or make you think again about it. It cannot replace human judgment. If you don't know how your system is working, then looking at graphs will not generally help you to understand it, since data needs a model or a theory to interpret it.

6.9 Patterns and Anomalies

The patterns revealed in the graphs produced by **cfenvgraph** can give us humans a gut impression of how a computer is being used by its users, and how the resources of the system are being consumed, over a time scale of several weeks. This information is useful for understanding performance in broad terms. If we see a large load at certain times of the day or the week, for instance, we could use this information to alter the configuration of resources at certain times to cope better with the load.

To organize your understanding about the system, examine the following checklist on each host in your organization:

- ❖ *Which measurements are predictable, i.e., which have a clear trend with small variations?* These are stable and predictable features, a sign that things are efficient and under control.
- ❖ *Which measurements are dominated by uncertainty, i.e., have large error bars with no visible pattern?* This occurs if the resource usage is only sporadic and irregular. It could be because resource usage is so low that you cannot see any pattern. The resource is not playing any role in your organization.
- ❖ *Which measurements have top-heavy distributions?* This might signal a resource that is being throttled by something, perhaps a performance limitation.
- ❖ *Which combinations of anomalies are most common in your system?* This will tell you the potential sources for instability and help you avoid unpleasant surprises in the future.

7. The Cfengine Management Process

Today businesses are more service-oriented than before, as testified to by the increasing interest in good practices such as ITIL (the IT Infrastructure Library) and NGOSS/eTOM (New Generation Operational Support Service/enhanced Telecom Operations Map) as championed by the Tele-Management Forum. These process models have certain requirements of practice. How does cfengine fit into this kind of process scenario?

7.1 Process Requirements

ITIL describes the following practices for good management:

- *Manager:* A service manager should be appointed to coordinate the management of services; for us, that means overseeing configuration management within the organization. The manager should determine the requirements of the "service client"—in our case, the configuration requirements of the organization. The configuration management workforce should be competent and well trained.

- *Documentation:* Policies should be documented. Performance should be reported. Security controls should be documented. Cfengine accomplishes these goals in a number of ways. The configuration language itself is designed to document the configuration policy to a large extent. You should also discuss the interaction of configuration options among different locations, processes, and hosts. The big picture is available only to a configuration manager or engineer. Clear documentation is a sign of good engineering—but, as we all know, knowing what is good documentation for future contingencies is harder than we think.

- *Service implementation:* The service provider promises to deliver the agreed service. It must allocate the appropriate funds and resources to make this happen. In this case, an organization has to install cfengine with an appropriate schedule for configuration management.

- *Monitoring:* The service provider promises to monitor its operation and seek continuous improvement of service provision. This includes testing of the service. "Monitoring" here does not refer to the performance or configuration monitoring of cfengine itself, but, rather, to the extent to which the current configuration policy is effective in driving the larger goals of the organization.

- *Change and revision control:* Service level agreements should adapt and be subject to revision control. In this case, this means that we should frequently review the policies, expectations, and cfengine schedules. As changes are made to the configuration policy, those changes should be documented and versioned using a revision control system. For example, the Subversion revision control system is both convenient and easy to use.

❖ *Continuity: I*TIL requires a plan for continuity of an enterprise. Redundancy must be established in order to provide the ability to continue to function in the absence of key dependencies and personnel. The critical dependencies in cfengine are, by design, minimal—as long as computers are running, cfengine should be running. Humans who understand the policy itself are a dependency, and one can interpret this requirement as the need for at least two staff members who understand the cfengine installation. The network might also be a dependency in some cases, although cfengine is designed to work under unreliable, partial communication conditions. If certain sources of data are required, e.g., servers for file copying, then failover servers can be provided.

7.2 Revision Control and Rollback

Cfengine does not version configurations internally, except to retain older version of files that are changed during copying and editing.

One idea that frightens system administrators about autonomic computing is that if a mistake is made (in policy or implementation), there is no clear way of "rolling back the change" to undo the damage. If you are thinking in this way, you are trapped by dangerous and costly preconceptions. System administrators often like to maintain the idea of version control on their system configurations, as they generally believe that they are in control of every aspect of their configurations. This is false.

There is a basic conflict between the idea of policy and version control. Policy-based configuration management is about control of final state, and the scope of the changes involved in reaching it. This approach to the state is not versioned. Either a system is correct, or it is incorrect.

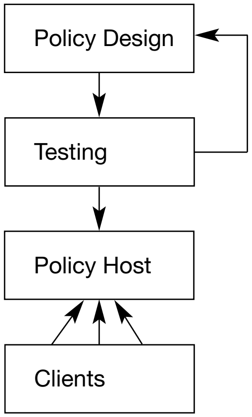

Figure 7.1: Elements of Revision Control

We wish to caution readers: just because you undo the last changes you made on a computer, this does not mean that you will get back to the state you were in previously, because runtime (operational) changes and consequences are not necessarily reversed by a reversal of configuration.

So what do we recommend?

- Keep your source configuration files under a version control system like Subversion so that you can track changes in your own thinking.
- Test new versions before rolling them out.
- When tested, update the master policy source with the new version.
- If, for whatever reason, the new policy has problems, either modify the policy again or go back to a previous version. In each case, cfengine implements changes in a forward direction, converging toward its policy, even if the policy has been rolled back.

This view of rollback and versioning might be an unfamiliar way of thinking to you, but it avoids several problems. The approach we advocate has the following properties:

- It avoids uncontrolled effects from ad hoc undo operations.
- It avoids complete reinstallation, e.g. versioning from image backup.
- By leaving the changes to cfengine, you are certain that the end result is that which you wrote in your policy.

In this approach, you are encouraged to be forward-thinking, rather than taking a defensive, backing-off strategy. We think that anyone implementing configuration management should have sufficient expertise to be confident about their changes after testing.

Index

About the Authors

Mark Burgess is Professor of Network and System Administration at Oslo University College. He is the author of the popular configuration management software package cfengine and is the author of numerous books and papers on network and system administration.

Æleen Frisch has been a system administrator for over 25 years, tending a plethora of (successively) VMS, UNIX, Windows, Linux, and Macintosh systems over the years. She is also a writer, lecturer, teacher, marketing consultant, and occasional database programmer.