

Distributed resource administration using cfengine

Mark Burgess/R.Ralston
Faculty of Engineering
Oslo College
0254 Oslo, Norway

May 9, 2003

Abstract

We describe experiences and frequently used configuration idioms for simplifying data and system administration using the GNU site configuration tool cfengine. Key words: system and network administration, automation, script language.

Introduction

System administration and data administration are often regarded as two independent pursuits. Increasingly, we are seeing this distinction eroded in contemporary software systems. Networked hosts can be viewed simply as a distributed database of active and passive resources, where active resources include processes and well known services and passive resources are files and data. Usually these two go hand in hand. The integrated management of all such resources requires a common standard of configuration for all the components involved.

Cfengine is a language based tool for implementing mass configuration of networked hosts[1, 2, 3, 4, 5]. It has been significantly developed and will be developed further in the future. It is in use at hundreds of sites around the world on all major platforms. Cfengine's aim is to provide a framework for managing UNIX-like information systems from a common interface: to be able to centralize and specify the correct state of a site configuration from a single file or centralized set of files with a common syntax. Cfengine provides a uniform method of configuring distributed systems in which concepts and decisions are centralized, but the actual work of configuration is distributed across all participating hosts on the network. Recently similar ideas have appeared in, for example, ref. [12, 13].

Various attempts have been made previously to simplify the issue of network administration. The designers of many systems attempt to put a friendly face on the job by wrapping up standard tasks with shell scripts and a graphical user interface. HPUX's SAM[6] and solaris' Solstice[7] are examples. Such tools are mainly aimed at novices; experienced administrators prefer to save time and edit the system directly by hand or by script. ¹ In any event, these programs—which might be helpful on one system—fail ultimately as effective administration tools because they are geared to a single a single system type, or a single vendor's products. They are also 'one off' set-up tools with no provision for monitoring the state of that configuration continuously and reporting or fixing errors. Other tools have been conceived which perform a continuous monitoring of systems (Palantir for instance[8]), but these systems do not perform any correctional tasks: they are only early warning systems (they always require human intervention). Other programs fail to be sufficiently customizable or adaptable to individual needs, and therefore fail to cope with locally motivated deviations from the vendor's vision of how the system should be set up.

Effective system administration depends not only on having functional tools, but on a continuous thread of understanding of the system as a whole and in all of its parts. If a change is made to a part of the system without all responsible administrators being aware of it, new problems can be introduced out of ignorance of those changes and havoc can be the result. With cfengine we create a configuration database for all hosts; this documents the setup of all systems in a single file (or set of files), thus cfengine permits all changes to be tracked and examined at a later date. Moreover, a cfengine configuration is robust to system re-installation: a particular configuration can be restored at any time by simply running the configuration engine on the host concerned.

The biggest challenge in system administration is to create a stable environment in which multiple platforms coexist and inter-operate. Cfengine places the burden of configuration on the computer itself, replacing scripts with a unix-independent interface which is specifically designed for managing heterogeneous environments. Customization, at any level, is integrated into a global framework, using an object oriented, hierachical philosophy.

This paper is not an introduction to cfengine (see refs. [2, 5] for such an introduction); rather we describe the behaviour and performance of cfengine in a number of frequently encountered scenarios. Although we select specific examples, we hope readers understand that they are meant to be representative and thought-provoking rather than extensive or encyclopaedic. We hope that this will advance the effective use of cfengine as a system administration tool and prompt further discussion about information and systems management. We urge readers to always think carefully before copying any system administration idiom, from any source.

¹Another approach for Hewlett Packard systems was described in ref. [9].

General issues

Cfengine grew out of the need to control the accumulation of complex shell scripts used in the automation of key system maintenance at Oslo. There were very many scripts, written in shell and in perl, performing tasks such as file tidying, find-database updates, process checking and several other tasks. In a heterogeneous environment, shell-scripts work very poorly: shell commands have differing syntax across different operating systems, the locations and names of key files differ. In fact, the non-uniformity of unix was a major headache. Scripts were filled with tests to determine what kind of operating system they were being run on, to the point where they became so complicated an unreadable that no-one was quite sure what they did anymore. Other scripts were placed only on the systems where they were relevant, out of sight and out of mind. It quickly became clear that our dream solution would be to replace this proliferation of scripts by a single file containing everything to be checked on every host on the network. By defining a new language, this file could hide all of the tests by using classes (a generalized 'switch/case'syntax) to label operations and improve the readability greatly. The gradual refinement of this idea resulted in the present day cfengine.

The use of classes to make decisions is cfengine's greatest advantage over rival tools. Tasks are placed in classes which determine either the hosts on which they should be executed, or the time at which they should be executed. Host-classes are essentially labels which document the attributes of different systems. They might be based on the physical attributes of the machine, such as its operating system type, or on some human attributes, such as its geographical location or departmental affiliation. The basic principles of this scheme have been described before[2, 5]. Actions are placed in such classes and are performed only if they belong to one of the classes relevant to the host which executes the cfengine program. What this means is that, by placing actions in judiciously chosen classes, one can specify actions to be carried out on either individual machines or on arbitrary groups of machines which have a common feature relating them. Classes are defined in a number of ways:

- *automatically* as a result of certain characteristics of the environment in which the cfengine program is run, e.g. the operating system type of the host, name of the host and the day on which the script is run etc. Cfengine senses its runtime environment and switches on these classes.
- *implicitly* by making the host a member of a named group of hosts, which then constitutes a class with the same name as the group. This is useful for specifying tasks to be performed on machines with a cultural or human connection, such as those belonging to a specific department at a university.
- *explicitly* by defining an identifier to be a defined class in the `control`

part of a cfengine program. This is useful for switching on and off certain tasks at run-time and is used in connection with the manual definition of classes listed below.

- *temporarily* by using identifiers in the action-sequence of the cfengine program. These are then defined only for the duration of the specified action. This technique is used to filter out certain tasks for specific situations, to achieve a finer control over the order of execution of multiple tasks.
- *manually* by using a command line option to cfengine which either defines or undefines a specified identifier as a class. This is also used to switch specific tasks on or off using runtime options.

The use of classes, as opposed to `if..then..else` decisions is particularly efficient since cfengine executes actions in bulk: instructions are not usually executed linearly from the start to the end of the file, rather cfengine defaults to execute all tasks of a particular type in one go. The ordering is preserved only within a particular type of task. The justification for this approach is that the readability of the cfengine program can be maximized in this way, using a very natural syntax. In the few cases where fine control is required, it can be achieved using the class mechanisms provided, but at the expense of the simplicity of the configuration. Creating an effective configuration for network resources amounts to placing configuration actions in appropriate classes. Some general guidelines for assigning classes may be observed:

1. There is no practical limit or penalty for the number of classes one may define and use so this feature can be exploited fully. Scripts are made clearer by liberally using meaningful class names. It cannot be emphasized enough that readability is one of the key reasons for creating cfengine in this area: computer programs are not merely monologues to a computer, but are also an important means of communication between humans. This should be reflected in cfengine programs.
2. Classes may depend on other classes, either implicitly or explicitly. This makes it possible to refer to all hosts in a given list except for a list of exceptions. Aliases can be made for commonly referred to hosts, such as the mailserver or the NIS server, so that they can be referred to by meaningful names and altered easily at a later stage.
3. Cfengine scripts can be divided into a number of files so that actions may be maintained as separate resources to be imported or included by any script, but the main idea is to have the same basic script or set of scripts for every host and to keep it in the same centralized place.
4. Classes may be combined into compound classes using logical operators. Thus hosts with complex properties may be pin-pointed easily.

5. Time consuming actions can be specifically labelled and switched off for quick-checks.

Information management on the network relies largely on a few key activities. In the following sections we discuss useful cfengine idioms for dealing with frequently performed tasks.

System and software installation

Cfengine may be used as a tool for software installation and for adapting brand new hosts to local requirements. Installation builds on a number of methods:

- Creating directories with appropriate ownership and permissions,
- Copying files from a master source and fixing their permissions, like the unix `install` program,
- Linking files and file-trees from a master source temporarily, or permanently, using both relative links and absolute links,

At the time of writing, cfengine lacks an explicit mechanism for remote copying from a network server, but this will be corrected in version 1.4.0. This may currently be simulated by NFS-mounting a server filesystem on the local host and copying the appropriate files with `copy`.

Symbolic and hard links are the canonical way of setting up a structured system, and allow data and software to remain in neat, self-contained packages, while being integrated into the whole. Links are used to make file trees appear in locations other than the directory in which they really lie and allow files to appear in several contexts without unnecessarily maintaining copies. Links can be soft, hard, absolute or relative. The default is to produce absolute symbolic links. Single links are trivially created using cfengine with a line of the following form:

`links:`

```
/path/link -> /srcpath/srcobject
```

The links are not just created, but also transparently checked for correctness. Multiple links are easily arranged for package integration. For example, to link the children of a given destination directory to the files and directories in a named source directory, one uses the syntax:

`links:`

```
destinationdir +> sourcedir
```

Note that this makes a link for every child in the named directory rather than making one link to the entire directory. This enables an entire directory of files to appear in a different place without having to make physical copies of the files. It also allows one to make a linked image of a file tree which in which some files are locally overridden by real files, whilst most are just linked to a source directory. Although some administrators undoubtedly feel that an excess of symbolic links will eventually lead to problems as files disappear, leaving links which point to non-existent files, cfengine can be made to perform link garbage collection, so this is not a problem in practice. An example is described under system maintenance, below.

More advanced linking schemes are required for mirroring source trees. For instance, it is possible to recursively link a file tree to a master source tree, so that each file is mirrored by a symbolic link, but sub-directories are actually created with the appropriate permissions.

links:

```
destination +> source recurse=inf
```

This is a method which is sometimes used in software administration in order to make several near-identical trees of files with custom additions. So-called link-trees are often used to build software for multiple platforms on the same disk space. In software installations it is advantageous to link most files to a master tree, but to copy others: customizable configuration files, for instance, should be copied rather than linked so that they can be made unique for the tree. This kind of setup may be handled by specifying a list of patterns which are to be copied rather than linked:

links:

```
destination +> source r=inf copy=*.ini copy=*.conf
```

The converse is also possible, i.e. to specify a file tree to be recursively copied, linking certain files matching a list of patterns:

copy:

```
source dest=destination r=inf link=lib*.so link=lib*.a
```

The effect is the same, but simplicity of expression might be greater in this juxtaposed viewpoint, and this is a key aim of cfengine.

Various path-type variables could be simplified, for example by symbolically linking all of the files in a list of directories to a single common directory.

```
/all-libs +> $(LD_LIBRARY_PATH)
```

Lists are detected by the presence of the list separator in the variable string — usually a list is separated by colons. Cfengine expands the composite command into a separate instruction for every item in the list. In the case of name conflicts, where several programs have the same name, one may arrange for one of the links to take precedence. Normally cfengine warns about such conflicts, but the `action=silent` switch allows this to be overridden. In practice few will need to use such as solution; the flexibility of links makes this more powerful than the shell's path feature however, since one may override a link with a file in any directory, without being restricted by the ordering of the path list in every case. This feature is not limited to the path variable, it may be used with any variable containing a list, and has been used in some of Hewlett Packard's software installation scripts where link trees need to be created.

Mountable resources

Mountable resources may be accessed either by statically mounting by editing `/etc/fstab` or its equivalent, or by dynamically mounting using the automounter. Cfengine can deal with both of these models. The issue one faces here is to edit the various configuration files controlling the mounters so as to mount only the remote filesystems which are relevant to the specific host concerned. The automounter does this automatically and can trivially be set up with the help of cfengine, but some sites prefer to use static mounting and this requires a selection procedure since some automounters require the Network Information Service, which is often problematical. We shall not belabour the point, but simply note that cfengine's ability to edit textfiles enables it to effortlessly set up mounting using both static and dynamical models, taking care of individual needs and efficiency.

System maintenance

What makes cfengine superior to many system administration tools is the converging semantics of its operation. Cfengine works by first checking the state of a host and then correcting or warning about flaws, so it can be run any number of times without risk of creating double links or adding two lines to a text file instead of only one. Once a host has reached the defined state of configuration, there is no question that re-runs of the engine will cause damage to that configuration. Cfengine semantics makes program reruns safe to everything except willfully coded do-undo contradictions (which only waste CPU cycles) and careless do-anyway editing commands. System maintenance consists of nothing more strenuous than running cfengine at suitable periodic intervals.

Maintenance typically involves the following issues:

- Checking the ownership and permission of files,

- Checking the correctness of symbolic links,
- Removing links which point to non-existent files,
- Updating certain files by copying from a master version, including `/etc/motd`
- Updating system databases (`find`, `locate`, `catman` etc),
- Checking for the existence or absence of named processes.

Let us consider these in turn.

Permissions

Files need to be checked for ownership and permissions on a regular basis. Often busy administrators can forget to change the ownership of files downloaded by `ftp`, so that such files lie on the system with a user id which either corresponds to no user on the local system, or worse, to a user who has no right to the files. Moreover, programs like `ftp` on some systems can leave files writable to the world (mode `666`), if `inetd` is restarted from the shell, since `inetd` and, in turn `in.ftpd` inherit `umask` from their parent shell. It is also annoyingly common that the ‘make install’ procedure in program packages leaves files or directories with permissions which make the installed program inaccessible to users.

Here is an example of how one might check and entire filesystem. The state of the `/usr/local`, installed software filesystem may be checked and corrected using the following idiom.

files:

```
AllBinaryServers.bigjob:
    /usr/local mode=o-w recurse=inf
                owner=root,bin group=0,1,2,3,4,5,6,7,staff
                links=tidy action=fixall
```

The `recurse` option indicates that `cfengine` is to start checking the file tree at the directory `/usr/local` and it to descend recursively into all subdirectories (unless they are on another device or are joined by symbolic links, in which case further options must be set). The `mode` option indicates that the 002-bit should be zeroed for any files (This could also be written `mode=-002`.) In other words, files which are writable to ‘others’ (the world) are made safe by removing the write permission. The ownership of a file is set to `root`, if it is not already `root`. While checking the file tree, `cfengine` is always on the lookout for `setuid-root`

and `setgid-root` programs. It keeps a list of such programs and issues a warning when a previously unregistered file is discovered.

A numerical list of group identifiers is used because group names are incompatible between different unices. If a file's group ownership is not one of those in the list, it will be set to the first value in the list, namely zero. Care should therefore be taken to ensure that all the required groups are covered here. Certain programs may cease to work if their group ownership is changed. The action `fixall` tells `cfengine` to not merely warn about problems, but to silently fix them. While testing a particular script it is wise to change this to `warnall` (the default) to see what `cfengine` is likely to do to the files.

The exceptional treatment of key files and directories is made possible by an ignore list. Files, directories and wildcard patterns may be ignored from all searches by adding them to a classified `ignore` list:

`ignore:`

```
specialhost::
```

```
    /usr/local/tmp
    !*          # emacs lock files
```

The ignored files may then be handled explicitly in a separate `files` action:

`files:`

```
specialhosts::
```

```
    /usr/local/tmp mode=1777 action=fixdirs
```

In the process of parsing the filesystem, `cfengine` makes a note of any `setuid` or `setgid` root programs. This includes any which might inadvertently be created as a result of change of ownership. It builds a list of these programs and checks whether any new `setuid/setgid` root programs appear. If so, a warning is generated which cannot be ignored. Finally, the option `links=tidy` tells `cfengine` that if it finds any symbolic links which point to non-existent files, they should be removed. This feature should be used with caution on user filesystems, since many users make links to filesystems which exist only on a particular host.

File imaging and copying

Another important idiom in system maintenance is the installation of up to date configuration files: the copying of master data from a central server. Many of the Network Information Service's (NIS) functions can be emulated in this way, for instance. `Cfengine` can update files from a master file, either by using a time

comparison or using a checksum comparison (the default is to use ctime). To update the password file from a master source (for networks not using NIS), one would write:

copy:

```
PasswdHost::  
  
    /etc/passwd dest=$(masterfiles)/etc/passwd mode=0444 owner=root  
  
!PasswdHost::  
  
    $(masterfiles)/passwd dest=/etc/passwd mode=0444 owner=root
```

Clearly the source and destination hosts must have the same password file format in this example. In a forthcoming version, it will be possible to side-step the intermediate file-repository and specify remote copying from a server. Copying of multiple files is trivial, using either a recursive copy idiom

copy:

```
# Copy directory contents to a max depth of 2 subdirs  
  
sourcedir dest=destdir recurse=2
```

or by iterating over an implied list. For instance, to distribute a basic shell setup for all users from a master configuration (or to install for new users), one could write

copy:

```
$(masterfiles)/cshrc.master dest=home/.cshrc
```

The use of the special variable `home` tells `cfengine` to iterate over the home-directories of all users. Thus, this simple line is an effective way to make sure that users always have a shell configuration. If a user were to accidentally delete the configuration file, `cfengine` would reinstall it. Any modifications users made would not be overwritten because of the time-stamp comparison. The same could conceivably be done with other key setup files, such as `.xsession`, `.fvwmrc`, provided the master versions remained static so that users' changes were not overwritten except in the case of an emergency. If a user accidentally

deleted, say, `.xsession` this would mean that he/she could never log in via the xdm login interface; adding an update of this file into the cfengine system configuration would fix this problem automatically for the user, even if the network administrator were at home or on vacation.

Editing

More subtle maintenance can be performed by *editing* files—clearly a feature to be used with some discretion. Critics looking for faults in cfengine often express concern about file editing. Many would rather construct enormous `sed` and `awk` scripts, piping through several processes which always rewrite the target file, than to use cfengine's single-process, easy syntax, write-only-if-necessary policy. Mostly these concerns seem to reflect an unwillingness to change old habits. Cfengine can easily accomplish many things which would be quite horrendous to accomplish with scripts. An important change in the configuration of the system could make it necessary to edit all users' setup files and make certain replacements. For instance:

editfiles:

```
{ home/.xsession

ReplaceAll "mwm" With "fvwm"

BeginGroupIfNoLineMatching ".*LD_LIBRARY_PATH.*"
  # Find first non-comment line
  LocateLineMatching "^[^#].*"
  InsertLine "setenv LD_LIBRARY_PATH /local/X11R6/lib"
EndGroup
}
```

Again, the `home` directive tells cfengine to iterate over all users' home directories.

Another use for file editing, is to update and distribute the message of the day file. Some administrators like to simply copy a standard file to `/etc/motd` on a regular basis. This means that details specific to a particular host cannot be handled individually. In particular any kernel messages about the OS release are lost. A slight improvement over this is to edit the file specifically for each host, using a configuration module which we shall call `cf.motd`. This file is imported into a larger cfengine configuration and it is assumed that the `editfiles` action will be run there.

```
#####
#
# cf.motd
#
```

```

# This file is used to set the message of the day file on
# each host
#
#####

control:

#
# This points to the file containing general text
#

masterfile = ( /iu/nexus/local/iu/motd-master )

editfiles:

{ /etc/motd

BeginGroupIfFileIsNewer "$(masterfile)"
EmptyEntireFilePlease
InsertFile "$(masterfile)"
PrependIfNoSuchLine "This system is running $(class):$(arch)"
EndGroup
}

linux::

{ /etc/motd

AppendIfNoSuchLine "Special message for linux users"
}

```

The script works by comparing the modification times (ctime) of the file against a master version. If the masterfile is older than /etc/motd, no editing is performed. However, as soon as this masterfile timestamp changes, cfengine empties the contents of the file and replaces it with i) a messages indicating the architecture and system type of the current host and (ii) a standard message file. Additional messages can be added for specific architectures or groups. An example for linux is shown.

Database maintenance

A regular task in system maintenance is to update the databases which are used to search for data quickly. These include the fast-find or GNU-locate database, and the whereis databases which are updated by the catman program. This is where cfengine's ability to execute controlled shell commands comes into its own.

shellcommands:

Hr00::

```
"$(gnu)/lib/locate/updatedb"
```

solaris.Saturday.Hr00::

```
"/usr/bin/catman -M /local/man"
```

```
"/usr/bin/catman -M /local/gnu/man"
```

Notice that some commands need to be operating system specific, owing varying syntax and location of files. Also, while it would probably do no harm (except perhaps in wear and tear on users' patience) to update these databases several times a day, it is normally not desirable to do so. Such shell commands should be singled out if you are planning to run cfengine several times a day, either by using a special class which is only defined once a day such as Hr00, or by running separate programs at different times, or perhaps by singling out a special day of the week. Actions may also be restricted to specific days of the week. See the section on Execution Strategies for more details about these points.

Garbage collection

Files

The number of junk files on networked systems seems to grow at an ever-increasing rate. Complex user-interfaces which make use of temporary files, combined with programs which cache network data (e.g. netscape) fill up disks with temporary files which many users are probably not even aware of. Moreover, compilation by-products like '.o' or '.dvi' files are often left lying around thoughtlessly, or because users do not understand them. Such files need to be removed forcibly in order to avoid disk overflows. Tidying strategies are a part of system policy and one should probably distinguish between experienced and inexperienced users here: users who understand all of their files might keep them for a reason. A simple idiom for tidying the temporary directories is the following.

tidy:

/tmp/	pat=*	r=inf	age=1
/var/tmp	pat=*	r=inf	age=1
/	pat=core	r=1	age=0

The tidy function deletes files matching the specified pattern only if they have not been *accessed* (by default) within the specified time, thus the first two lines recursively empty the temporary directories of files which have not been used for more than a day. The last line checks the root directory and its children (to a depth of one sub-directory) and removes core files, regardless of their age.

The temporary directory items should normally be supplemented with an ignore item:

```
ignore:
```

```
.X11
```

which prevents cfengine from deleting window manager data in `.X11` subdirectories and causing irritating error messages to users.

It is not only users who inadvertently sabotage the system by allowing large files to accumulate. Many operating systems are programmed to do this automatically. If disk space is tight, then steadily growing system log files can be a real problem. For example, the World Wide Web server logs grow for each access of the server. On a busy server, it would not take long to clock up a log file of many megabytes.

Cfengine implements the standard unix idiom of 'rotating' log files and also allows files to be emptied. File rotation refers to the behaviour exhibited by syslog and by the messages logs and a number of other programs, namely that old log files are renamed by adding consecutive numbers to them. The file `messages` is thus renamed `messages.1`, allowing the messages file to start afresh. The old `messages.1` file is renamed `messages.2` and so on. After a certain maximum number is reached, the files 'fall off the end' and disappear forever. This behaviour allows old logs to be examined for a limited period, so that administrators can trace problems back in time.

Not all programs rotate their logs automatically, but cfengine can be made to enforce this with an idiom of the following form:

```
disable:
```

```
Sunday::
```

```
    /var/log/mylogfile rotate=4
```

In this example, the log file is rotated (keeping a maximum of four files) each time this rule is executed. In this case, the rule is only executed on Sundays, which is often sufficient to keep the size of slowly growing log files to a minimum. One thing to look out for with this method is programs which keep their logfiles open at all times (such as the `httpd` daemon. If the files are rotated, this causes

a 'stale file handle' condition and the daemon is unable to write to the new file without being restarted. To restart the daemon one could write

processes:

```
"httpd" signal=term restart "/local/web/httpd -d /local/web"
```

though, incidentally, not that not all programs detach themselves properly from their parent process. If this is the case then such a restart command will hang the cfengine process. There is a better method for clearing old log files however, since many old log files are simply not of any great interest. One can use the `rotate=empty` idiom. In this case the log file in question is simply emptied without any file rotation taking place. This does not create a stale file handle, since the file is never substituted by another, and the daemon will continue to write to the file. If necessary the file can be backed up with a `copy` command first. Here are some examples:

disable:

```
WWWServer::
```

```
    /local/httpd_1.4/logs/access_log rotate=empty
```

```
solaris::
```

```
    /var/lp/logs/lpsched rotate=empty
```

```
solaris.Wednesday|solaris.Sunday::
```

```
    /var/adm/wtmpx      rotate=empty
```

```
    /var/adm/wtmp       rotate=empty
```

The irony is that many operating systems expect hosts to be rebooted in order to clear old logs. Thus stable hosts are penalized for being up too long.

Some administrators keep a large dummy file on each partition which can be deleted in order to perform an emergency save while other measures can be considered. Nowadays, it is often sufficient to delete all netscape cache files to free a considerable number of megabytes in one fell swoop. The following code fragment will usually clear many megabytes quickly.

tidy:

```
home/.netscape pattern=*      age=0
```

```
home          pattern=core    age=0
```

```
home          pattern=*.o      age=0

# etc
```

Processes

On multi-user hosts, garbage collection also needs to be performed on processes. Programs such as `pine` and `elm` are known to hang and consume vast amounts of CPU time, as does Solaris' `lpNet` printer daemon and `netscape`. The same applies to many network clients. One (primitive but effective) solution to this problem is to limit the amount of CPU time per process using the 'set limit' feature of the C-shell in a way which can be overridden by users, if need be. For instance, one could edit users' setup files to place a limit of one hour on all processes.

editfiles:

```
{ home/.cshrc # or .user_cshrc

AppendIfNoSuchLine "limit cputime 3600"
}
```

This method would work for users who freely wished to comply with a CPU limit, but others wishing to run long jobs could easily find this approach a nuisance. The `cfengine` process action can be used to deal with such cases instead. The trick here is to search for processes using a regular expression which matches processes which have clocked up too much CPU time, and then send them the kill or terminate signal.

processes:

```
Hr00::

"pine" signal=kill
```

`Cfengine` matches the output of `ps aux`, or its equivalent, which contains the name `pine` if `cfengine` is executed when the local time is between 00:00 and 00:59 and kills the relevant processes. A more advanced regular expression might be constructed to single out only those processes which have been running for more than a reasonable length of time.

Processes which fork uncontrollably, or even excessive thrashing on an overloaded system, sometimes make it necessary to kill processes which do not belong to root. This can be done with the aid of a regular expression which matches lines in `ps aux`, or its equivalent, which contain the string 'root' zero times.

processes:

```
"\(\root\)\{0\}" signal=term
```

The backslashes are required to escape the special regular expression operators.

System upgrades

System upgrades are performed by copying and perhaps relinking packages and file trees, either from a network server or from some mountable medium (CD, tape or network filesystem). Cfengine may be used to upgrade software from mountable filesystems using the regular `copy` action, thereby effecting remote copy. This works by piggy-backing off the remote sharing systems built into the mount protocol. Updates may be performed either by copying using a `ctime-comparison`, or using a `checksum comparison`. File trees can be upgraded recursively using the standard idiom for recursive copying:

copy:

```
source dest=destination recurse=inf
      type=checksum define=done
```

shellcommands:

```
done::
```

```
"/local/bin/post-process-script"
```

The syntax `define=done` tells us that the class `done` will be switched on only if the file is copied. This feature allows us to label a number of actions which will be performed only if copying was actually done. This might include unpacking a tar file during an automatic upgrade, or updating a hashed password database after copying the masterfile.

One of the reasons that cfengine does not yet implement a remote copying mechanism is that several solutions already exist. One of these is *rdist*[10]. Rdist works by pushing files from a central server out to a number of clients, forcibly updating old files using a time comparison. This method has, in the past, been a good solution in many instances since time can be saved by sending update packets to several hosts simultaneously. Amongst the disadvantages with this approach are that the pusher must have `rcmd` authorization in order to update a client. This means that all clients must open themselves to the distribution server, which is a potential security hazard.

Other solutions to the problem have been created. Hewlett Packard's *ninstall* system works in reverse, allowing users to access the central server to download

the latest version of the operating system and supporting packages. This ‘pull’ method has many advantages over *rdist*, since it allows each system to decide when is the right time to upgrade. Also it only requires policing the security of the server; the individual clients do not have to lend root authority to other systems. Linux and FreeBSD also have similar package upgrade mechanisms. These package upgrade schemes are not quite generic file copying mechanisms.

The system envisaged for cfengine is some kind of intelligent pull mechanism based on pure remote copy. A post processing facility will allow package files to be unpacked and installed, so that even ftp archives will be collectable automatically. The aim is to create a flexible and efficient system. The configuration engine itself is not (and should not be) a server; it should at most contain client code for connecting to an appropriate server, A separate program will be used for the server.

Recent advances in networking technology present a number of interesting possibilities. For example, the Adaptive Communications Environment (ACE)[16] forms a set of wrappers which provide a standardized interface for network services. This environment is in many ways superior to models based around *inetd*, for instance. ACE makes transparent multithreading of the server and admits the possibility of file servers on non-unix-like systems such as NT. On the negative side, the ACE environment is a monster to compile.

Configuration of services: editing

Configuration of databases and system services is generally accomplished by text-editing. Basic text editing functions were originally introduced into cfengine as a simple and incomprensive convenience. It became quickly apparent that it would be impossible to quench users’ insatiable thirst for an ever greater number of more complex and intelligent editing commands. What is important for cfengine’s editing commands is that editing semantics are reproducible and non-contradictory under multiple runs.

Text editing is straightforward, so we shall not belabour the point. Instead we illustrate with an example. In the our example, cfengine is used to make sure that the services file recognizes a number of important services which are not normally present in vendors’ services files.

Similarly, new services must be added to the configuration file for the internet daemon. In this example we add the bootp server for booting Xterminals.

```
XBootServer::
{ /etc/inetd.conf
AppendIfNoSuchLine "bootp dgram udp wait root /local/bin/bootpd bootpd -i -d"
}
```

After such an edit, the daemon needs to be sent the hangup signal, forcing it to reread the configuration file. This can also be automated with the following action:

```
processes:
```

```
    "inetd" signal=hup
```

Security

Cfengine is not an instrument for administering security, but it can be used together with other tools to implement reasonable checks. Security consciousness includes monitoring access permissions on files, checking for the presence or absence of key files, checking the integrity of key files against a master version with the help of a checksum, monitoring which processes are running and being aware of set userid programs which could compromise the system. Several crackers have been interrupted and detected at Oslo by cfengine's process monitoring facility. Security also means data security, regular backups and protecting the system from crippling situations such as full disks and amok processes.

The first of these issues, file permissions, has already been discussed above. As a further example we could mention the setup and monitoring of an anonymous ftp server as an example where permissions need to be kept under tight control.

Cfengine can make comparisons based on MD5 checksums[15]. Checksum comparisons are useful for binary data. Since MD5 checksum cannot be faked, if any user, by whatever means, managed to replace a program binary with a Trojan horse, cfengine would determine this and replace the file with a new copy from the master file.

A known bug in versions of solaris before 2.5 is that the /tmp directory is installed without the sticky bit set. This means that any user can delete any other users' files. A simple line fixes this.

```
files:
```

```
    solaris::
```

```
        /tmp mode=1777 action=fixdirs
```

Cfengine's automatic setuid-root and setgid-root monitor keeps a log of previously known files and alerts the administrator to any new files which appear. A regular check of user areas can detect the arrival of potential security risks as well as ensuring that users' files are not writable to the whole world. (Many

users do not understand the `umask` variable, and some are not even aware of file permissions.)

files:

```
home mode=o-w action=fixall
```

As a service to users we could police their files and remove world writable flags added in ignorance. Unfortunately, it is difficult to distinguish between users who do this out of ignorance and those who do it on purpose.

In addition to checking for permissions and ownership, we must police the state of certain files. Some vendors maintain the bizarre habit of installing their operating systems with a NIS '+' symbol in the file `/etc/hosts.equiv`. This file is part of the `rcmd` authorization mechanism: hosts which are listed here may log in to the current host without supplying a password. When the plus symbol is added, one effectively adds every host in the current NIS domain to this file. This is clearly a gaping security hole and may be fixed as follows:

editfiles:

```
{ /etc/hosts.equiv
DeleteLinesMatching "+"
}
```

A better solution for a more secure environment is to simply forbid this file to exist at all.

disable:

```
/etc/hosts.equiv
```

This causes the file to be renamed or removed to a dead-file repository, out of harm's way. The permissions on the file are also changed to 400 (read only for the owner). This makes the `disable` action a suitable way to disable certain dangerous programs, such as `setuid` root programs which contain security loopholes. (CERT advisory messages on compromised software often include recommendations for disabling certain files in this fashion.) Other security files which can be monitored in a similar way include `/etc/hosts.lpd` and the `cron.allow/deny` files.

Some files *must* be in place. A surprise at Oslo occurred when a disk glitch eliminated the kernel configuration file `/etc/system` from a solaris host. Without this file, the kernel will not fork any processes and thus will not even run the startup scripts which boot the system.

files:

```
solaris::
```

```
/etc/system o=root g=root m=644 action=touch
```

Had the problem been noticed, it would have been a simple matter to touch the file before rebooting the system. This has now been added to the cfengine configuration.

Several system-crackers have been apprehended at Oslo by monitoring for a small list of unwanted processes. For example:

processes:

```
"eggdrop"  signal=kill  
"ping"     signal=kill
```

The `eggdrop` program (a TCL IRC client/server which gives a limited shell access to outsiders, and is often disguised by renaming it 'csh' or 'vi') and the `ping` have proven to be crackers' hallmarks. Cfengine has alerted us to several crackers (who gain access to the system through accounts with weak passwords) by informing that it has killed around a hundred ping processes, directed at a few hosts in what is presumably a ping attack against the remote systems. Normal users do not run repeating ping processes, and even if they should do so once or twice, it would be no catastrophe to have them killed by cfengine—thus the above ping example is a worthwhile addition to a site configuration.

Some system administrators are of the opinion that the finger program constitutes a security leak and wish to disable the finger service. This may be dealt with for all hosts, site-wide, with a fragment of the form:

editfiles:

```
{ /etc/inetd.conf  
  
SetCommentStart "#"  
SetCommentEnd ""  
CommentLinesContaining "finger"  
}
```

The HUP signal can then subsequently be sent to `inetd`, to force it to reread the new configuration.

Execution strategies

In the preceding sections we have described the functionality of the configuration engine and its language interface. A separate issue, but no less important, is how and when the engine should be made to do its work. How often should we check the state of each host on the system?

There are two principal ways in which `cfengine` is run at Oslo:

1. from a cron script, using the wrapper program supplied with the distribution. Any output generated by `cfengine` is mailed to the system administrator. The main configuration file has two modes of operation: a default mode in which everything is done, and an ‘hourly’ mode in which only a subset of lightweight operations is performed.
2. by remote shell, interactively, in response to a specific problem.

The first of these methods is reminiscent of the daily, weekly and monthly scripts run by the system in FreeBSD unix. A notable difference is that, while FreeBSD always mails information on each pass, `cfengine` reports only about problems which it has not been authorized to fix automatically.

The frequency of runs depends very much on local considerations. At Oslo, difficulties with NIS mean that password file distribution is performed by file-copying. This means that an hourly run is required to keep these passwords up to date. Large jobs are reserved for weekend evenings when the system is rarely loaded. Both modes of execution may be combined in a single program using classes to distinguish hourly actions from the remainder.

Experience reveals both strengths and weaknesses in the cron model. The strength of the model is that there is a strong probability that `cfengine` will be run, even when the system cannot be contacted remotely. Occasionally, for example, it becomes impossible to log onto SVR4 hosts, because of a ‘protocol error’. This usually means that an important disk partition is full. If `cfengine` is running, it can tidy junk files and create sufficient space to permit remote login, whereupon the problem can be examined by a human.

On the other hand, this timed method is not sufficient for all cases. Special circumstances might require the immediate reconfiguration of some or all hosts, for instance, to “push” password files to all hosts after an alteration. Moreover, the cron method assumes that `cfengine` will always be run. If for some reason cron can not run `cfengine`, perhaps because the host was down, then clearly no mail will be sent by `cfengine` to report the problem. Thus the flaw in the “silence means OK” policy is that severe problems cannot be detected. A further problem can occur if very long `cfengine` scripts are run too often: there is then

the risk that the scripts will take so long that the previous run will not be finished before the next run is attempted. Cfengine's pid-lock should prevent this from happening, but this can result in spurious error messages being mailed to the administrator.

Another issue is how the output of cfengine should be collected from the various hosts on the network. As of today, the only official solution is to mail the data on a per-host basis to an address which is defined in the configuration file. While this method works well enough, a more pleasing user interface would allow cfengine to log its output by date and time in a central logging server.

In the future a service-based method of running cfengine will be tested, using a daemon which handles logging of output and remote connection. In this model, it will be possible to set up a central control panel which polls machines at regular intervals, running cfengine on each system. In this way it is possible to monitor whether all hosts are running, and if so, whether they are running cfengine. The output from each host is then fed into a central logging server where it can be rationalized and sorted. This model also allows immediate distribution of a configuration file, using an encrypted key system of authentication.

An alternative would be to use the httpd daemon, a system already in widespread use. This has both advantages and disadvantages. Amongst the advantages are that this service is well known to administrators and contains some of the functionality required to do the job already. Amongst the disadvantages are security issues, such as the fact that httpd runs as user nobody. Cfengine would have to be run setuid root in order to have control over the system concerned. This makes it very difficult to implement a secure distribution of configuration files.

We believe that the best currently available strategy for configuring hosts mixes both static cron-based scheduling and a service based interface with host monitoring. A more advanced solution will be presented in a future paper.

Conclusion

Cfengine provides an adaptable language interface for collecting and automating many data and system administration tasks into a central site-wide set of configuration files. In this paper we have catalogued a number of helpful idioms for how cfengine may be used to automate system administration. Cfengine is now in widespread use at internet sites around the world. The software is obtainable from any GNU ftp server and up to the minute information may be obtained from the web site[3, 4].

Future work on cfengine will include an intelligent remote copying mechanism and a more sophisticated launch[18] and logging system, linkable to the world wide web. It is possible that cfengine could be combined with the Rscan WWW based system[17] to achieve this end. At Oslo we are working on a se-

cure method of running cfengine remotely which will partially replace the cron method. This will have the advantage of combining a 'ping' check on every host with a method for instantly updating the cfengine configuration on every host. Our collaboration at Hewlett-Packard has led to the use of cfengine as a software configuration tool. A port of cfengine to the NT operating system would be a major contribution to network administration. This work is in progress. We hope to see many more uses for cfengine in the future and urge users to follow the developments on the cfengine web service.

M.B. would like to thank David Masterson for his many helpful contributions to the cfengine project.

References

- [1] M. Burgess, Invited talk at the HEPix meeting at Saclay 1994. <http://wwwcn.cern.ch/hepix/meetings/saclay94.html>
- [2] M. Burgess, *A site configuration engine*. Computing systems **8**, volume 3, 309 (1995)
- [3] The cfengine web site: <http://www.iu.hioslo.no/~mark/cfengine>
- [4] Two news groups have been created: gnu.cfengine.bug, for bug reports and problems, and gnu.cfengine.help, for general discussion. The latter is unmoderated.
- [5] CFENGINE, documentation, Free Software Foundation 1995/6. This is also available online at the web site in ref. [3].
- [6] HPUX system administration manual.
- [7] Solstice Administration Guide, Sun Microsystems.
- [8] Palantir was a project run by the University of Oslo centre for Information technology (USIT). Details can be obtained from palantirusit.uio.no. and <http://www.palantir.uio.no>. I am informed that this project is now terminated.
- [9] M. E. Shaddock, M.C. Mitchell and H.E. Harrison, *How to upgrade 1500 workstations on Saturday, and still have time to Mow the Yard on Sunday*, in proceedings of the ninth systems administration conference LISA, (SAGE/USENIX), 1995
- [10] M.A. Cooper, *Overhauling Rdist for the '90s*, in proceedings of the sixth systems administration conference LISA, (SAGE/USENIX), 1992.
- [11] X. Gittler, W.P. Moore and J. Rambhaskar, *Morgan Stanley's Aurora system: designing a next generation global*

- [12] C. Hogan, *Decentralising distributed systems administration*, in proceedings of the ninth systems administration conference LISA, (SAGE/USENIX), 1995.
- [13] Host factory, URL: <http://www.wv.com>.
- [14] P.W. Osel and W. Gänsheimer, *Opendist - Incremental Software Distribution*, in proceedings of the ninth systems administration conference LISA, (SAGE/USENIX), 1995.
- [15] The MD5 algorithm is described in RFC 1321.
- [16] Publications, references and other information about the ACE library may be obtained from Douglas Schmidt's web site at: <http://http://siesta.cs.wustl.edu/~schmidt/ACE.html>.
- [17] N. Sammons, *Multi-platform interrogation and reporting with Rscan*, in proceedings of the ninth systems administration conference LISA, (SAGE/USENIX), 1995.
- [18] kondo@takaoka-nc.ac.jp, private communication.