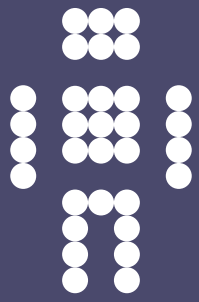
CFEngine



CFEngine 3 Tutorial

A CFEngine AS workbook

Mark Burgess @ CFEngine AS

Table of Contents

1 System automation

1.1 Managing diverse and challenging environments seamlessly and invisibly

The future is never far away. Our dream of a future in which smart computing devices are embedded into the very fabric of our environment has crept slowly into being. Today, smart operating systems like Linux and Windows are used on embedded devices and mobile phones. Mark Weiser of Xerox PARC once wrote:

> "*The most profound technologies are those that disappear. They weave themselves into the fabric of every day life until they are indistinguishable from it.*"

Today many are talking about Cloud Computing as another manifestation of this dream, in which computing service is not only everywhere, but nowhere – or more correctly, spread out across the planet in data centers, instead of our offices and homes. This is one aspect of making computing into something we take for granted. At the foundations of any such technology are the tools required to implement mass configuration with surgical precision. CFEngine is such a tool.

CFEngine was designed to enable scalable configuration management, for the whole system life-cycle, in any kind of environment. Almost every other system for configuration assumes that there will be a reliable network in place and that changes will be pushed out top-down from an authoritative node. Those systems are useless in environments like

- Mobile systems with partial or unreliable connectivity (e.g. a submarine).
- Systems where bandwidths are very low (e.g. a satellite or space probe).
- Systems where computing power is very low (e.g. ad hoc sensors or kitchen appliances).

CFEngine does not need reliable infrastructure. It works opportunistically in almost any environment, using few resources. It has few software dependencies. So, not only does it work in all of the traditional fixed-plan scenarios, but it is capable of working in totally ad hoc deployment: an temporary incident room, a submarine drifting on and off line, a satellite or a robot explorer.

One could argue 'well I don't need that kind of system, because my network is reliable'. However, your network is not as reliable as you think, and mobility is an increasingly important topic. Even with a very strong redundant network, the services that support the network can be paralyzed by any of a number of failed dependencies or mishaps. It is crucial in a modern pervasive environment that systems remain available, fault tolerant and as far as possible independent of external requirements. This is how to build scalable and reliable services.

> CFEngine works in all the places you think it should, and all the new places you haven't even thought of yet. How do we know? Because it is based on almost 20 years of careful research and experience.

1.2 Managing expectations - a theory of promises

One of the hardest things in management is to make everyone aware of their roles and tasks, and to be able to rely on others to do the same. *Trust* is an economic time-saver. If you can't trust you have to verify, and that is expensive.

To improve trust we make promises. A promise is the documentation of an intention to act or behave in some manner. This is what we need to learn to trust systems, no matter whether they are machines or humans.

One CFEngine user once said to me, that the thing that had helped him the most in deploying CFEngine was its design based around voluntary cooperation. "Our main problems were not technical but political – getting everyone to agree in all of our departments around the world". This was because, for all the technology, it is people who make the decisions and people need to feel that the system is empowering rather than disempowering them.

CFEngine works on a simple notion of promises. Everything in CFEngine can be thought of as a promise to be kept by different resources in the system.

Combining promises with patterns to describe where and when promises should apply is what CFEngine is all about.

1.3 Why automation?

Humans are good at making decisions and awful at reliable implementation. Machines are pitiful at making decisions and very good at reliable implementation. It makes sense to let each side do the job that they are good at.

The main problem in managing systems is a loss of self-discipline. Discipline does not imply that order have to be barked from a central command. It only requires that every part of the system knows its job and carries is out seamlessly and flawlessly.

Skilled workers tend to think that it is enough to be smart. In fact this is wrong: smart people tend to be problem solvers and will happily solve the same problem many times, wasting time and effort. Moreover, human intervention is often based on panic and lack of understanding so every time someone logs onto a system by hand, they jeopardize everyone's understanding of the system. Only the self-discipline of stable procedures leads to predictability.

Ad hoc changes are bad because:

- Others have no idea what happened.
- There is no record of changes or intentions.
- A scar is left from the change.

People often rile against automation saying that it dehumanizes their work. In fact the opposite is true: forcing humans to do the work of machines, in repetitive and reliable ways is what dehumanizes people. The only way to make progress with a bad habit is to recognize it and be willing to abandon the habit.

1.4 Scaling up

In the past, the only way to scale up system numbers was to make all systems identical. This is no longer true.

In the late 1960s journalist and futurist Alvin Toffler sketched a pretty compelling vision of the western world and its post-industrial future. His book Future Shock, which appeared in 1970, was really a reaction to the cold-war fears about a communist industrial state in which mass production made everything and everyone identical and indistinguishable. His book was

really a rebuttal to all those who argued that industrialization and mass production implied that everything had to be exactly the same, and I recommend reading it - it is very well written and has many lessons for us today. But from his rather long diatribe, I wrote down a single sentence which for me sums up the lesson that we have failed to learn:

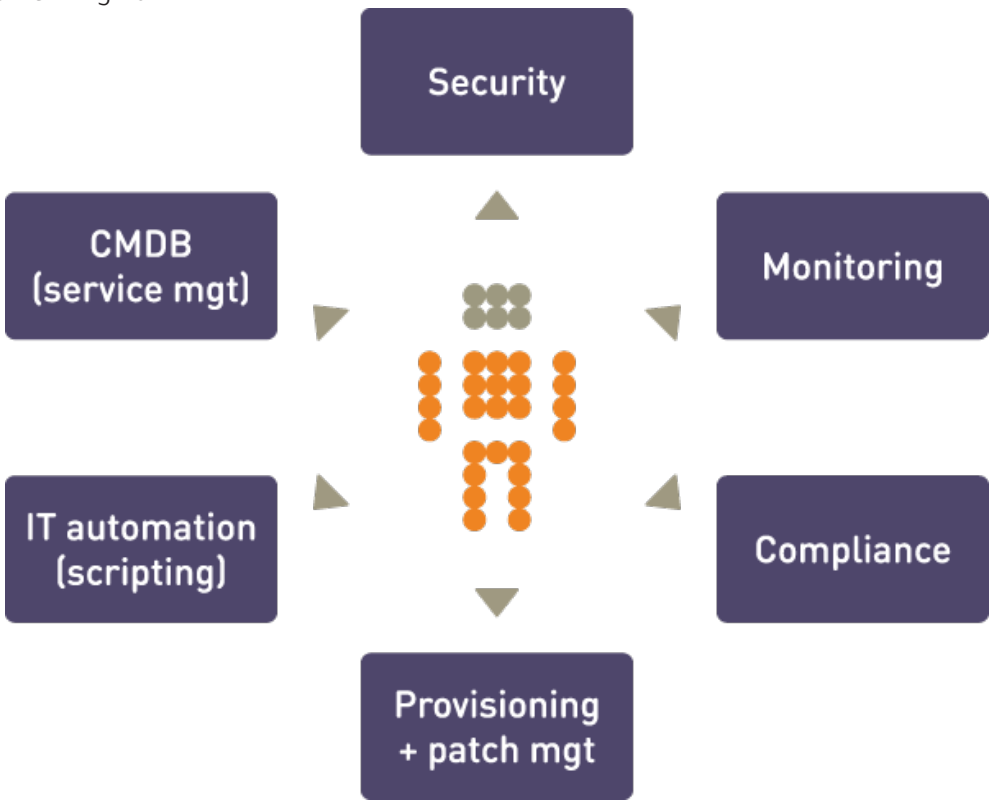
> "*As technology becomes more sophisticated, the cost of introducing variations declines.*"

In other words, any half-decent technology for mass production would help us to be more sophisticated and multifaceted, not less. In an age when you can get business cards printed on demand from an ATM at the airport, and personalized coffee mugs in the blink of an eye, there is no reason to perpetuate the myth that massive infrastructure requires monolithic replication, and yet people still do. Network engineers do, and system administrators do. They even say that this is essential for scalability.

The importance of Toffler's message was that the economics of mass production are not at odds with the economics of adaptation, but 40 years later, we are still re-learning that lesson.

1.5 How do *you* view CFEngine?

CFEngine is a framework. It is not so complex, but it is certainly extensive. Often when trying to describe CFEngine, it seems that there is too much to tell and it is hard to convey in a simple way what the software can do. The picture below shows a few ways in which you can think of CFEngine.



For many users, CFEngine is simply a configuration tool – i.e. software for deploying and patching systems according to a policy. Policy is described using promises – indeed, every statement in CFEngine 3 is a promise to be kept at some time or location. More than this,

however, CFEngine is not like most automation tools that 'roll out' an image of some software once and hope for the best. Every promise that you make in CFEngine is continuously verified and maintained. It is not a one-off operation, but an encapsulated process that repairs itself should anything deviate from the policy.

That clearly places CFEngine in the realm of automation, which often begs the question: so it's just another scripting language? Certainly CFEngine contains a powerful scripting language, but it is not like any other. CFEngine is not a low level language like Perl, Python or Ruby; it is a language of promises, in which you express very high level intentions about the system and the inner details figure out the algorithms needed to implement the result. We'll return to this below.

For many, CFEngine is a tool for implementing security hardening procedures on systems, and monitoring them continuously thereafter. This is certainly a major application area. CFEngine has a reputation for being reliable and secure. That is because its basic design is secure: it is not possible to send information about policy to CFEngine from outside the system. If access has been granted, it is only possible to send a few simple protocol requests of limited length to the server. This makes the design safer than most firewalls. Most servers fail security tests because it is possible to send data to them.

The ability to describe almost any kind of policy for a system means that we can suggest promises that a system should make and comply with. Thus CFEngine can also be thought of as a compliance engine. It is easily used to comply with frameworks like SOX, 'EUROSOX' (the EU 8th Data Directive), ITIL and standards like ISO 17799, ISO 20000, etc.

Finally, although CFEngine was not initially conceived for monitoring, it contains one of the most flexible and lightweight monitoring engines around. You can extract data about system configuration, usage, resources and log data and turn this into readable reports. CFEngine's ability to discover and extract information about the system, combined with its reporting means that you can turn the system into a simple Configuration Management Database. In the Community edition, monitoring is a zero-touch background process. With CFEngine commercial extensions, there is almost no limit to the kind of monitoring promises you can make, and without the embarrassing resource spikes that many monitoring systems produce.

Above all, CFEngine is aimed to promote human understanding of complex processes. Its promises are easily documentable using comments that the system remembers and reminds us about in error reporting. It hides irrelevant and transitory details of implementation so that the *intentions* behind the promises are highlighted for all to see. This means that the knowledge of your organization can be encoded into the CFEngine language.

WHY DOES KNOWLEDGE MATTER? There are two reasons: the first is that technical descriptions are hard to remember. You might understand your configuration decisions when you are writing them, but a few months later when something goes wrong, you will probably have forgotten what you were thinking. That costs you time and effort to diagnose. The second reason is that organizations are fragile to the loss of those individuals who code policy. If they leave, often there is no one left who can understand or fix the system. Only with proper documentation is it possible to immunize against loss.

2 The components of CFEngine

CFEngine comprises a number of components. In this chapter we'll consider how to build them and what they are for.

2.1 Installation

To install CFEngine, you will need a few packages. You require:

OpenSSL Open source Secure Sockets Layer for encryption.
URL: http://www.openssl.org

Tokyo Cabinet (version 1.4.42 or later)
Lightweight flat-file database system.
URL: http://fallabs.com/tokyocabinet/

PCRE Perl Compatible Regular Expression library.
URL: http://www.pcre.org/

On Windows machines, you need to install the basic Cygwin DLL from http://www.cygwin.com in order to build CFEngine Community.

Additional functionality (some of which is available only in commercial extensions) also becomes available if other libraries are present, e.g. OpenLDAP, client libraries for MySQL and PostgreSQL, etc. It is possible to run CFEngine without these, but related functionality will be missing.

Unless you have purchased ready-to-run binaries, or are using a package distribution, you will need to compile CFEngine. For this you will also need a build environment with `gcc`, `flex`, and `bison`.

The preferred method of installation is then

```
tar zxf CFEngine-x.x.x.tar.gz
cd CFEngine-x.x.x
./configure
make
make install
```

This results in binaries being installed in '`/var/cfengine/bin`'.

2.2 The work directory

CFEngine keeps a work space directory for its own use. The default location for this is '`/var/cfengine`' when run as the root user, and `~/.cfagent` for other users.

```
/var/cfengine
/var/cfengine/bin
/var/cfengine/inputs
/var/cfengine/outputs
```

A trusted cache of the input files must now be maintained in the '`inputs`' subdirectory. When CFEngine is invoked by the scheduler, it expects to read only from this directory. It is up to the user to keep this cache updated, on each host (this is arranged by the default configuration files).

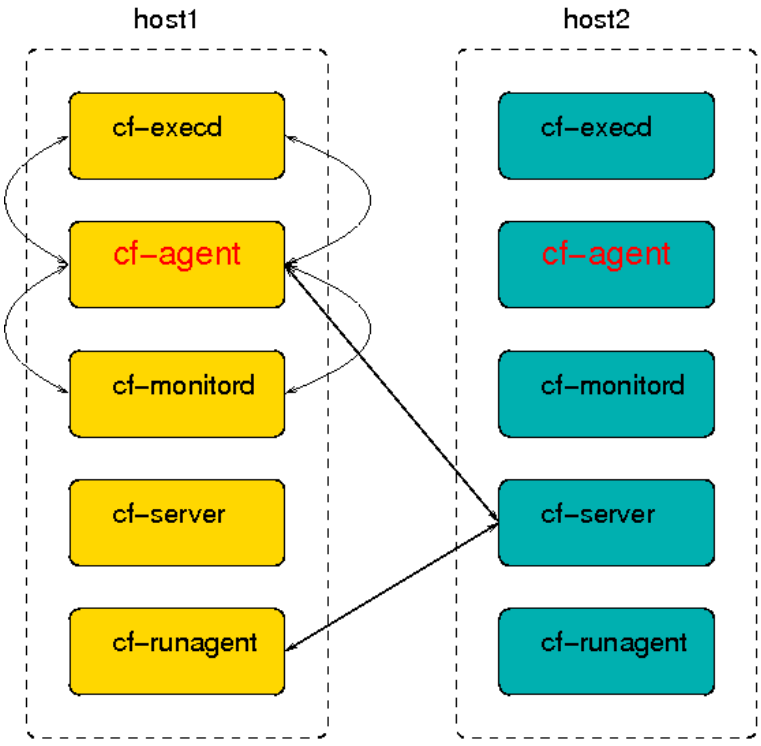
Unlike CFEngine 2, CFEngine 3 does not recognize the `CF-INPUTS` environment variable.

The '`outputs`' directory is now a record of spooled run-reports. These are often mailed to the administrator by `cf-execd`, or can be copied to another central location and viewed in an alternative browser.

2.3 The players

A CFEngine system is something like an orchestra. It is composed of any number of computers (players), each of which has its own copy of the music and knows what to play. It might or might not have a conductor to help coordinate the individual parts – that's up to you.

CFEngine's software agents run on each individual computer but can communicate if they need to, as depicted the figure below. This means you don't have to arrange risky login credentials to run your network – and if something goes wrong with the communications network, CFEngine is where it needs to be to repair or protect the system during the outage.



If the network is not working, CFEngine just skips these parts and continues with what it can do. It is fault tolerant and opportunistic.

cf-promises

The promise verifier and compiler. This is used to pre-check a set of configuration promises before attempting to execute.

cf-agent

This is the instigator of change. The agent is the part of CFEngine that manipulates system resources.

cf-serverd

The server is able to share files and receive requests to execute existing policy on an individual machine. It is not possible to send (push) new information to CFEngine from outside.

cf-execd

This is a scheduling daemon (which can either supplement or replace `cron`). It also works as a wrapper, executing and collecting the output of `cf-agent` and E-mailing it if necessary to a system account.

cf-runagent

This is a helper program that can talk to `cf-serverd` and request that it execute `cf-agent` with its existing policy. It can thus be used to simulate a push of changes to CFEngine hosts, if their policy includes that they check for updates.

cf-report

This generates summary and other reports in a variety of formats for export or integration with other systems.

cf-know

This agent can generate an ISO standard Topic Map from a number of promises about system knowledge. It is used for rendering documentation as a 'semantic web'.

2.4 About the CFEngine architecture

This section explains how CFEngine will operate autonomously in a network, under your guidance. If your site is large (thousands of servers) you should spend some time discussing with CFEngine experts how to tune this description to your environment as *scale* requires you to have more infrastructure, and a potentially more complicated configuration. The essence of any CFEngine deployment is the same.

There are four commonly cited phases in managing systems, summarized as follows:

- Build
- Deploy
- Manage
- Audit

These separate phases originate with a model of system management based on transactional changes. CFEngine's conception of management is somewhat different, as transaction processing is not a good model for system management, but we can use this template to see how CFEngine works differently.

Build A system is based on a number of decisions and resources that need to be 'built' before they can be implemented. Building the trusted foundations of a system is the key to guiding its development. You don't need to decide every detail, just enough to build trust and predictability into your system.

In CFEngine, what you build is a template of proposed promises for the machines in an organization such that, if the machines all make and keep these promises,

the system will function seamlessly as planned. This is how it works in a human organization, and this is how is works for computers too.

Deploy Deploying really means implementing the policy that was already decided. In transaction systems, one tries to push out changes one by one, hence 'deploying' the decision. In CFEngine you simply publish your policy (in CFEngine parlance these are 'promise proposals') and the machines see the new proposals and can adjust accordingly. Each machine runs an agent that is capable of implementing policies and maintaining them over time without further assistance.

Manage Once a decision is made, unplanned events will occur. Such incidents traditionally set off alarms and humans rush to make new transactions to repair them. In CFEngine, the autonomous agent manages the system, and you only have to deal with rare events that cannot be dealt with automatically.

Audit In traditional configuration systems, the outcome is far from clear after a one-shot transaction, so one audits the system to determine to discover what actually happened. In CFEngine, changes are not just initiated once, but locally audited and maintained. Decision outcomes are assured by design in CFEngine and maintained automatically, so the main worry is managing conflicting intentions. Users can sit back and examine regular reports of compliance generated by the agents, without having to arrange for new 'roll out' transactions.

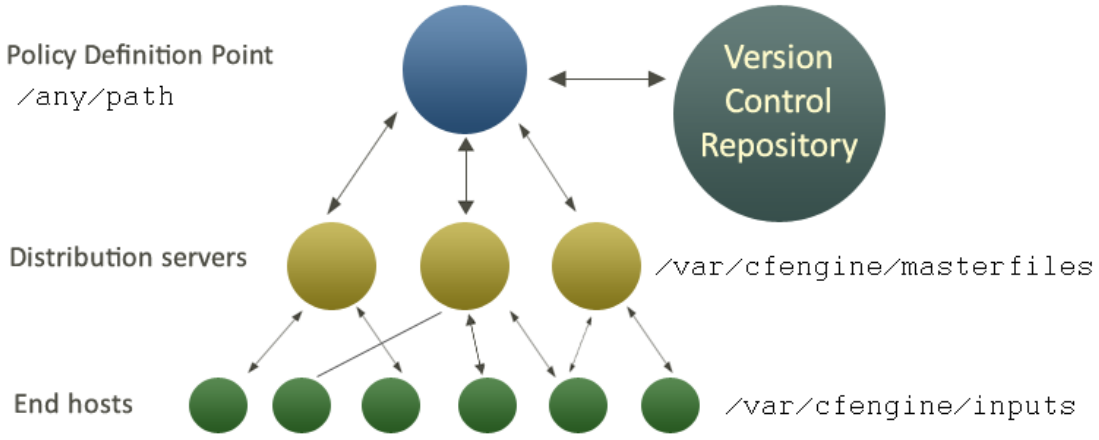
ROLL-OUT and ROLL-BACK? You should not think of CFEngine as a roll-out system, i.e. one that attempts to force out absolute changes and perhaps reverse them in case of error. Roll-out and roll-back are theoretically flawed concepts that only sometimes work in practice. With CFEngine, you publish a sequences of policy revisions, always moving forward (because like it or not, time only goes in one direction). All of the desired-state changes are managed locally by each individual computer, and continuously repaired to ensure on-going compliance with policy.

2.5 The policy decision flow

CFEngine does not make absolute choices for you, like other tools. Almost everything about its behaviour is matter of policy and can be changed. However, a structure for use, like the following, is recommended (see the following figure).

In order to keep operations as simple as possible, CFEngine maintains a private working directory on each machine referred to in documentation as WORKDIR and in policy by the variable `$(sys.workdir)`. By default, this is located at '`/var/cfengine`' or '`C:\var\CFEngine`'. It contains everything CFEngine needs to run.

The figure below shows how decisions flow through the parts of a system.



- It makes sense to have a single point of coordination. Decisions are therefore usually made in a single location (the Policy Definition Point). The history of decisions and changes can be tracked by a version control system of your choice (e.g. Subversion, CVS, etc.).
- Decisions are made by editing CFEngine's policy file '`promises.cf`' (or one of its included sub-files). This process is carried out off-line.
- Once decisions have been formalized and coded, this new policy is copied *manually* (a human decision) to a *decision distribution point*, which by default is located in the directory '`/var/cfengine/masterfiles`' on all policy distribution servers.

  In this introduction, we shall assume that there is only one central policy distribution server, a specially-appointed server which is referred to simple as the `policy server`.
- Every client machine contacts the policy server and downloads these updates. The policy server can be replicated if the number of clients is very large, but we shall assume here that there is only one policy server.

Once a client machine has a copy of the policy, it extracts only those promise proposals that are relevant to it, and implements any changes without human assistance. This is how CFEngine manages change.

> *WHY DO THIS? CFEngine tries to minimize dependencies by decoupling processes. By following this pull-based architecture, CFEngine will tolerate network outages and will recover from deployment errors easily. By placing the burden of responsibility for decision at the top, and for implementation at the bottom, we avoid needless fragility and keep two independent quality assurance processes apart.*

2.6 Getting started with the Community Edition

The quickest way to get started with CFEngine is to download and install binary packages, available from `http://cfengine.com/inside/myspace`. Installing and bootstrapping these is a trivial operation, putting you two steps away from testing your first CFEngine policies.

Note: There is a bug in Community 3.3.0 where the default policy files are not copied to `/var/cfengine/masterfiles` upon bootstrap. Workaround consists in manually copying these files before bootstrapping:

```
cp /var/cfengine/share/CoreBase/*.cf /var/cfengine/masterfiles/
```

This will be corrected in a bugfix release coming soon.

This procedure applies to all hosts, but you should bootstrap the hub (policy server) first. Find the hostname or IP address of the hub, here we assume the address is '123.456.789.123' (do not bootstrap with a localhost address).

```
host# /var/cfengine/bin/cf-agent --bootstrap --policy-server 123.456.789.123
```

CFEngine will output diagnostic information upon bootstrap (written to command line and syslog; cf-agent will also return a value: ERROR: 1, SUCCESS: 0). Error messages will be displayed if bootstrapping failed, pursue these to get and indication of what went wrong and correct accordingly. If all is well you should see the following in the output:

```
-> Bootstrap to 123.456.789.123 completed successfully
```

As an alternative to installing binary packages, the following steps outline the procedure when you have built from source (available from http://cfengine.com/source_code). You will then need to copy the distributed policy files that were installed in '`/var/cfengine/share/CoreBase/`' to a policy distribution point, like this:

1. Decide on your policy server.
2. Become root or Administrator on that server.
3. Create the policy source directory and populate the cache:

```
host# /var/cfengine/bin/cf-key
host# cp /var/cfengine/share/CoreBase/*.cf /var/cfengine/masterfiles/
host# /var/cfengine/bin/cf-agent --bootstrap --policy-server 123.456.789.123
```

CFEngine should be up and running on your system after the bootstrap. It will copy its default policy files into '`/var/cfengine/masterfiles`' on the hub (policy server) provided that the directory is empty (fresh install). This directory is the definition point for your policy, meaning that clients will contact the hub and copy these files to their '`/var/cfengine/inputs`' directory. Note that the policy hub also works as a client against itself, so it too will copy from '`/var/cfengine/masterfiles`' to '`/var/cfengine/inputs`'. You should browse the files in '`/var/cfengine/masterfiles`' to see what they contain, and perhaps make some alterations to adapt to your environment.

To see which CFEngine processes are running:

```
host# ps waux | grep cf-
```

Note: If you have manually configured a different location for the CFEngine work directory, you will need to adapt these lines above to replace '`/var/ CFEngine`' with the path you have configured; e.g. Debian based packages feel that '`/var/ lib/CFEngine`' is the right location for this.

3 Bodies and bundles

To emphasize the fact that CFEngine is not an imperative programming language, and to keep closely to the nomenclature of Promise Theory, CFEngine uses two concepts throughout: bundles and bodies.

3.1 Bodies

Promises are the fundamental statements in CFEngine. Promises are the policy atoms. If there is no promise, nothing happens.

However, promises can become quite complicated and readability becomes an issue, so it is useful to have a way of breaking them down into independent components. The structure of a promise is this:

Promiser This is the object that formally makes the promise. It is always the *affected object*, since objects can only make promises about their own state or behavior in CFEngine.

Promisee (optional)
This is a possible stakeholder, someone who is interested in the outcome of the promise. It is used as documentation, and it is used for reasoning in the commercial CFEngine product.

Promise body
Everything else about a promise is defined in the body of the promise. We use this word in the sense of 'body of a contract' or the 'body of a document' (like `<body>`) tags in HTML, for example.

A promise body is a list of declarations of the following form:

```
CFEngine_attribute_type => user_defined_value or template
```

3.1.1 Body parts

The CFEngine reserved word `body` is used to define *parameterized templates* for bodies to hide the details of complex promise specifications. For complex body lists, you must fill in a body declaration as an 'attachment' to the promise, e.g.

```
files:

  "/tmp/promiser"        # Promiser

    perms => myexample;  # The body is just one line,
                         # but needs an attachment
```

The attachment is declared like this, with a 'type' that matches the left hand side of the declaration in the promise:

```
body perms myexample
{
mode => "644";
owners => { "mark", "sarah", "angel" };
```

```
groups => { "users", "sysadmins", "mythical_beasts" };
}
```

The structure is this:

```
  promiser

    LVALUE => RVALUE

..

body LVALUE RVALUE
{
LVALUE => RVALUE;
LVALUE => RVALUE;
}
```

Another way of looking at it is this:

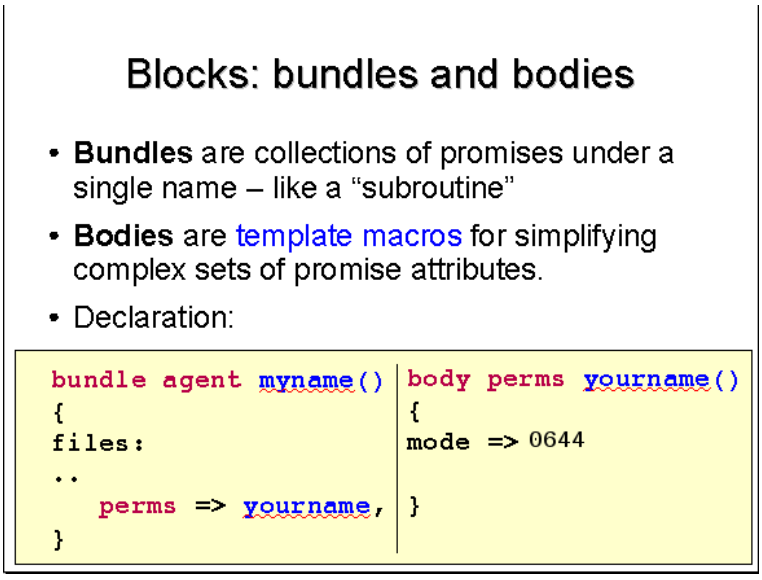
```
  promiser

    CFEngine_word => user_defined_value

..

 body CFEngine_word user_defined_value
 {
 CFEngine_word => user_defined_value;
 CFEngine_word => user_defined_value;
 ...
 }
```

Body attachments are required items. You cannot choose to put the attachments in-line. This is a lesson that was learned from CFEngine 2. Readability is quickly lost if too many details are placed in-line.



3.1.2 Control bodies

Some promises in CFEngine are implicit. They are hard-coded into the program. For example, the fact that CFEngine looks for a number of files to read and will execute them in a sequence is hard coded. You cannot change this. However, you can change the behavior of such promises by setting control parameters. These are formally parts of the 'promise body' for these hard-coded promises, so we use the body structure to set them. Each agent has a special body whose name is `control`; e.g.

```
body agent control
{
bundlesequence => { "test" };
}
```

3.2 Bundles

A bundle is a simple concept. A bundle is merely a collection of promises in a 'sub-routine-like' container. The purpose of bundles is to allow you greater flexibility to break down the contents of your policies and give them names. Bundles also allow you to re-use promise code by parameterizing it.

Like bodies, bundles also have 'types'. Bundles belong to the agent that is used to keep the promises in the bundle. So `cf-agent` has bundles declared as

```
bundle agent my_name
{
}
```

The `cf-serverd` program has bundles declared as:

```
bundle server my_name
{
}
```

and so on.

3.2.1 Bundle scope

Variables and classes defined inside bundles are not directly visible outside. All variables in CFEngine are globally accessible, however if you refer to a variable by '`$(unqualified)`', then it is assumed to belong to the current bundle. To access any other (scalar) variable, you must qualify the name using the name of the bundle in which it is defined: '`$(bundle_name.qualified)`'.

Some promise types, like `var`, `classes` may be made by any agent. These are called `common` promises. Bundles of type `common` are special. They may contain common promises. Classes defined in common bundles have global scope.

3.3 A simple syntax pattern

The syntax of CFEngine follows a simple pattern in all cases. Once you have learned this pattern, it will make sense anywhere in the program. The figure below illustrates this pattern. Some words are reserved by CFEngine, and are used as types or categories for talking about promises. Other words (in blue) are to be defined by you. Look at the examples and try to identify these patterns yourself.

The pattern

Cfengine word		User defined word
What is it?	What's it for?	What's its name?
bundle	agent server	myname
body	perms signals	yourname

4 How to execute and test a CFEngine policy

You do not need root privilege to use CFEngine. Most experiments can be safely tested as an ordinary user. You should spend some time experimenting with small examples before setting out to configure a system. To do that you should log onto your system as a regular unprivileged user and set up:

```
host$ /var/cfengine/bin/cf-key
host$ cp /var/cfengine/bin/cf-* ~/.cfagent/bin
```

CFEngine wants to see copies of its binaries in its work directory. For a regular user this lies in '`~/.cfagent`' rather than '`/var/cfengine`'. You should now be ready to go.

4.1 Hello world

Here is the simplest 'Hello world' program in CFEngine 3:

```
# Every policy must have a bundlesequence

body common control
{
bundlesequence  => { "test" };
}

#

bundle agent test
{
reports:              # This is a promise type

 cfengine_3::         # This is a class context (the promise will only
                      # be kept on a CFEngine_3 system)

  "Hello world";   # This is a simple promise (it generates a report
                   # that says "Hello world")
}
```

Type this in to a file, e.g. '`emacs ~/test.cf`'. Then check the syntax like this

```
/var/cfengine/bin/cf-promises -f ~/test.cf
```

If all is well there should be no output. Now execute as follows:

```
/var/cfengine/bin/cf-agent -f ~/test.cf
```

You should see this:

```
R: Hello world
```

The '`R:`' tells you this is the output from a report (as opposed to a log '`L:`', or the quoted output of some embedded program '`Q:`').

This is not a typical CFEngine program, primarily because CFEngine is not normally meant to print messages except in exceptional circumstances. As a starter however, it is reassuring to see some output.

If you repeat the command immediately nothing will happen. But if you wait a minute, it will work again. Run the command in verbose mode (use the `-v` or the `--verbose` switch) to see why:

```
/var/cfengine/bin/cf-agent --verbose -f ~/test.cf
```

Now you will see:

```
cf3>    ------------------------------------------------------------
cf3>    reports in bundle hello (1)
cf3>    ------------------------------------------------------------
cf3>
cf3>  XX Nothing promised here [lock.hello.reports..Hello_worl] (0/1
minutes elapsed)
cf3>
```

This tells you that CFEngine believes it is too soon to try to keep this promise again. The time it sets on this is determined by the `ifelapsed` parameter, which can be set individually for every promise. You can also ask CFEngine to ignore these locks using the '`-K`' option.

Before the 'Hello world' string, you see the class expression '`cfengine_3::`'. This is how CFEngine makes decisions. The promise to print the message will only apply if this condition is true. To see that this class is true for the execution, look at the verbose output from the command you just typed. You will see something like this:

```
Defined Classes = ( any verbose_mode Tuesday Hr08 Morning Min48
Min45_50 Q4 Hr08_Q4 Day7 July Yr2009 Lcycle_2 GMT_Hr6 linux atlas
undefined_domain 64_bit linux_2_6_27_23_0_1_default x86_64
linux_x86_64 linux_x86_64_2_6_27_23_0_1_default
linux_x86_64_2_6_27_23_0_1_default__1_SMP_2009_05_26_17_02_05__0400
compiled_on_linux_gnu localhost_localdomain localhost net_iface_lo
net_iface_wlan0 ipv4_192_168_1_100 ipv4_192_168_1 ipv4_192_168
ipv4_192 fe80__21c_bfff_fe6e_70ef CFEngine_3_0_2b4 CFEngine_3_0
CFEngine_3 SuSE lsb_compliant suse suse_n/a suse_11_1 suse_11
agent )
```

i.e. a list of all the currently defined classes. Any one of these classes (or a combination) could have been used to label the promise. That is the way CFEngine points to which promises will be kept in which scenarios.

A final thing to note: if you try to process this using the '`cf-promises -r`' command, you will see something like this:

```
atlas$ ~/LapTop/CFEngine3/trunk/src/cf-promises -r -f ~/test.cf
Summarizing promises as text to ~/test.cf.txt
Summarizing promises as html to ~/test.cf.html
```

The '`-r`' option produces a report. Examine the files produced:

```
cat ~/test.cf.txt
firefox ~/test.cf.html
```

You will see a summary of how CFEngine interprets the files, either in HTML or text. All the CFEngine components will produce debugging file with an expanded view when using this option (e.g. for the configuration file named '`promise_output_agent.h`', they will create the files '`promise_output_agent.html`' and '`promise_output_agent.txt`').

4.2 Checking a file

Type in the following example:

```
body common control
{
bundlesequence  => { "test" };
inputs => { "cfengine_stdlib.cf" };
}

bundle agent test
{
files:

# This is a throw-away comment, below is a full-bodied promise

"/tmp/testfile"                      # promiser

   comment => "This is for keeps...", # Live comment
    create => "true",                 # Constraint 1
     perms => m("612");               # Constraint 2, rw---x-w-

}
```

In the CFEngine Community Open Promise-Body Library (CFEngine_stdlib.cf) is a library of definitions that can be obtained from the CFEngine website. It should be included in the 'inputs' directory and input as above. Within that file, the template 'm' is defined:

```
# This is a trivial body template, which makes parameterizing
# the promise body tidier and re-usable

body perms m(x)
{
mode => "$(x)";
}
```

This example shows how additional attributes are added to the body of the promise. The right hand side of the perms declaration is a template which we have called 'm()', which uses a parameter. The template is defined below the bundle of promises that uses it, showing how we can create re-usable sets of parameters. In this case, the example is trivial, but we have barely begun. When things get more sophisticated, we shall hide a huge amount of detail in these parameters, thus keeping the main promise uncluttered and its intention clear.

In every 'promise constraint' of the form 'left-hand-side => right-hand-side', the left hand side is a CFEngine reserved word, and the right hand side is a decision you make, possibly expressed in terms of standard templates.

Now execute cf-agent with this promise:

```
host$ /var/cfengine/bin/cf-agent -f /tmp/test.cf -I
-> Object /tmp/testfile had permission 600, changed it to 612
```

```
host$ ls -l /tmp/testfile
-rw---x-w- 1 mark users 33 2009-06-30 06:06 /tmp/testfile
```

The '-I' flag tells CFEngine to 'inform' us about changes only. This provides a digestible amount of output that is more than the default (which is to only report un-fixable problems or explicit reports). We see that CFEngine creates the file as ordered, and sets the permissions appropriately. Now try to change the permissions:

```
host$ chmod 400 /tmp/testfile

host$ ls -l /tmp/testfile
-r-------- 1 mark users 33 2009-06-30 06:06 /tmp/testfile

host$ /var/cfengine/bin/cf-agent -f /tmp/test.cf -I
-> Object /tmp/testfile had permission 400, changed it to 612

host$ ls -l /tmp/testfile
-rw---x-w- 1 mark users 33 2009-06-30 06:06 /tmp/testfile
```

Once again, remember the comment about locking and `ifelapsed` from the previous example.

Notice that this promise does not have a class expression like `cfengine_3::`. The default class `any::` applies if nothing is stated, which means 'anytime, anyplace, anywhere'.

4.3 Changing a password

To change root password of a system, we need to edit a file. A file is a complex object – once open there is a new world of possible promises to make about its contents. CFEngine has bundles of promises that are specially for editing. Make a copy of a shadow file and copy it to '`/tmp`' so that you can play with it.

```
body common control
{
bundlesequence  => { "test" };
inputs => { "cfengine_stdlib.cf" };
}

bundle agent test
{
files:

"/tmp/shadow"
       comment => "Set the root password",
     edit_line => set_user_field("root",2,"xyajd673j.ajhfu");

}
```

This is all we need to see on first inspection to understand the promise that is being made.

4.4 The update bundle - provisioning

The default CFEngine configuration contains a bundle of promises that copies the CFEngine binaries into the cache directory and copies the policy files from the server into the default location. This example is for local copying from file to file on the filesystem. Later, when we set up a server component, you will be able to copy from a remote host. This is a simple example of system provisioning, with automated update.

```
bundle agent update
{
vars:

# A standard location for the source point
"master_location" string => "/var/cfengine/masterfiles";

files:

"/var/cfengine/inputs"

  comment => "Update the policy files from the master",
  perms => u_m("600"),
  copy_from => u_cp("$(master_location)","localhost"),
  depth_search => u_recurse("inf");
}
```

These promises contain several attributes in their bodies that we have not seen yet. The `copy_from` attribute tells CFEngine how to source (copy) a file from a master location. The `depth_search` tells it to search recursively through the sub-directories and their files.

Try changing the source files and executing the agent.

Again there are library reusable templates:

```
body perms u_m(p)
{
mode  => "$(p)";
}

#

body copy_from u_cp(from,server)
{
servers     => { "$(server)", "failover.example.org" };
source      => "$(from)";
compare     => "digest";
}

#
```

```
body depth_search u_recurse(d)
{
depth => "$(d)";
exclude_dirs => { "\.X11", ".*kde.*", "logs", "log" };
}
```

Here is an exercise: try using the reference manual to look up the elements in this example. See if you can understand all the parts.

4.5 Reporting

CFEngine contains a report generator called '`cf-report`'. It is configured using control parameters described in the next chapter. Try:

```
host$ /var/cfengine/bin/cf-report
host$ ls ~/.cfagent/reports
host$ mywebbrowser ~/.cfagent/reports/performance.html
```

Most of these reports will be blank at the start, until you have run CFEngine on some significant promises.

4.6 `cf-execd`

CFEngine contains a service for running the agent with its default configuration in '`WORKDIR/inputs/promises.cf`' called the exec-daemon. If you execute the binary directly it will go into the background and execute '`cf-agent`' every five minutes by default, with its default policy.

You can try running it in the foreground:

```
host$ /var/cfengine/bin/cf-execd -F
```

When you run CFEngine like this, any output that comes from CFEngine is collected and placed in '`WORKDIR/outputs`'. If you have configured an email address and your host is running an SMTP service, then it will be sent as email. To configure this you would add a control body to the '`promises.cf`' file

```
body executor control

{
splaytime => "1";
mailto => "cfengine_mail@example.org";
smtpserver => "localhost";
mailmaxlines => "30";
}
```

These other lines change different aspects of the hard-wired behavior of the executor, e.g. a load-balancing time delay before execution of the agent, a mail address, the name or IP address of an SMTP (mail) service, and the maximum number of lines of output to be included in any email sent.

You should start to see a pattern in the way CFEngine is configured. In the next chapter, we'll look at these general matters.

5 A simple crash course in concepts

5.1 Rules are promises

Everything in CFEngine 3 can be interpreted as a promise. Promises can be made about all kinds of different subjects, from file attributes, to the execution of commands, to access control decisions and knowledge relationships.

This simple but powerful idea allows a very practical uniformity in CFEngine syntax. There is only one grammatical form for statements in the language that you need to know and it looks generically like this:

```
type:

classes::

  "promiser" -> { "promisee1", "promisee2", ... }

     attribute_1 => value_1,
     attribute_2 => value_2,
     ...
     attribute_n => value_n;
```

We speak of a promiser (the abstract object making the promise), the promisee is the abstract object to whom the promise is made, and then there is a list of associations that we call the 'body' of the promise, which together with the promiser-type tells us what it is all about.

> The promiser is always the object affected by the promise.

Not all of these elements are necessary every time. Some promises contain a lot of implicit behavior. In other cases we might want to be much more explicit. For example, the simplest reports promise looks like this:

```
reports:

"hello world";
```

And the simplest commands promise looks like this

```
commands:

"/bin/echo hello world";
```

This promise has default attributes for everything except the 'promiser', i.e. the command string that promises to execute. A more complex promise contains many attributes:

```
# Promise type
files:

# promiser                  -> promisee (no curly braces needed if only one)
"/home/mark/tmp/test_plain" -> "system blue team",
```

```
      # attribute => value
          comment => "This comment follows the rule for knowledge  integration",
          perms   => owner("@(usernames)"),
          create  => "true";
```

The list of promisees is not used by CFEngine except for documentation, just as the comment attribute (which can be added to any promise) has no actual function other than to provide more information to the user in error tracing and auditing.

You see several kinds of object in this example. All literal strings (e.g. `"true"`) in CFEngine 3 must be quoted. This provides absolute consistency and makes type-checking easy and error-correction powerful. All function-like objects (e.g. `users("..")`) are either built-in special functions or parameterized templates which contain the 'meat' of the right hand side.

5.2 Control promises

Certain promises that CFEngine components make are hard-wired into their code. For example, the promise to email output to an appropriate address, or the promise to wait until a certain time has elapsed before checking a promise again (`ifelapsed`). Although these promises are hard-wired, their behavior can be changed. In CFEngine, behavior is always constrained by the promise body. Thus hard-wired behavior is altered by changing the control body for each. You can find these alterable parameters in the reference manual.

The most important bundle is the `common` bundle, that is read by all components of CFEngine. It contains the list of promise bundles that should be read in and examined for promise suggestions. From the '`promises.cf`' file:

```
body common control
{
bundlesequence  => {
                   "update",
                   "garbage_collection",
                   "main",
                   "cfengine"
                   };

inputs          => {
                   "update.cf",
                   "site.cf",
                   "library.cf"
                   };
}

#########################################################

body agent control
{
# if default runtime is 5 mins we need this for long jobs
```

```
ifelapsed => "15";
}

############################################################

body monitor control
{
forgetrate => "0.7";
histograms => "true";
}

############################################################

body executor control

{
splaytime => "1";
mailto => "cfengine_mail@example.org";
smtpserver => "localhost";
mailmaxlines => "30";
}

############################################################

body reporter control

{
reports => { "performance", "last_seen", "monitor_history" };
build_directory => "/tmp/nerves";
report_output => "html";
}

############################################################

body runagent control
{
hosts => {
        "127.0.0.1"
        # , "myhost.example.com:5308", ...
       };

}

############################################################

body server control
```

```
{
allowconnects          => { "127.0.0.1" , "::1" };
allowallconnects       => { "127.0.0.1" , "::1" };
trustkeysfrom          => { "127.0.0.1" , "::1" };

# Make updates and runs happen in one

cfruncommand           => "$(sys.workdir)/bin/cf-agent -f failsafe.cf &&
$(sys.workdir)/bin/cf-agent";
allowusers             => { "root" };
}
```

5.3 Variables

Variables (or "variable definitions") are also promises – the promise to represent their values. We can write these in any promise bundle. CFEngine recognizes two object types: scalars and lists (lists contain 0 or more objects), as well as three data-types (string, integer and real). Typing in CFEngine is dynamic, as in Perl and other scripting languages. Thus variables of any data-type may be used as strings.

5.3.1 Scalar variables

Scalar variables hold a single value. The are declared as follows:

```
bundle <type> name
{
vars:

"my_scalar" string => "String contents...";
    "my_int" int    => "1234";
   "my_real" real   => "567.89";

}
```

The '<*type*>' indicates that any kind of bundle applies here. Scalar variables are referenced by '`$(name)`' (or '`${name}`') and they represent a single value at a time.

- Scalars that are written without a context, e.g. '`$(myvar)`' are local to the current bundle.
- Scalars are globally available everywhere provided one uses the context to verify them e.g. '`$(context.myvar)`' may be written to access the variable 'myvar' in bundle 'context'.

5.3.2 List variables

List variables hold several values. The are declared as follows:

```
bundle <type> name
{
vars:

   "my_slist" slist => { "list", "of", "strings" };
   "my_ilist" ilist => { "1234", "5678" };
   "my_rlist" rlist => { "567.89" };
```

```
}
```

An entire list is referred to with the at symbol '@', but it does not usually make sense to use this reference in a string. For instance

```
reports:

  cfengine_3::

     "My list is @(my_slist)";
```

means nothing and cannot be expanded (it does not generate an error, but instead inserts the text @(my_slist) into the string); but if we use the scalar reference to a list variable, CFEngine will iterate over the values in the list essentially making this into a list of promises.

To summarize:

- Scalar references to *local* list variables imply iteration, e.g. suppose we have local list variable '`@(list)`', then the scalar '`$(list)`' implies an iteration over every value of the list.
- Lists can be passed in their entirety in any context where a list is expected as '`@(list)`'., e.g.

  ```
  vars:

  "longlist" slist => { @(shortlist), "plus", "plus" };

  "shortlist" slist => { "you", "me" };
  ```

  The declaration order does not matter – CFEngine will execute the promise to assign the variable '`@(shortlist)`' before the promise to assign the variable '`@(longlist)`'.
- Only local lists can be expanded directly. Thus '`$(list)`' can be expanded but not '`$(context.list)`'. Global list references have to be mapped into a local context if you want to use them for iteration. See the reference manual for more information.

5.3.3 Associative arrays

Associative array variables also hold several values. The are declared as follows:

```
bundle <type> name
{
vars:

   "switches[mellow]" int => "1";
   "switches[relaxed]" int => "1";
   "off_keys" slist => { "red", "grouchy", "coarse", "febrile" };
   "switches[$(off_keys)]" int => "0";

}
```

See the reference manual for information on the '`getindices`' function and other details of associative arrays.

5.4 Decisions

CFEngine decisions are made behind the scenes and the results of certain true/false propositions are cached in Booleans referred to as 'classes'. There are no if-then-else statements in CFEngine; all decisions are made with classes.

CFEngine runs on every computer individually and each time it wakes up the underlying generic agent platform discovers and classifies properties of the environment or context in which it runs. This information is effectively cached and may be used to make decisions about configuration.

Classes fall into hard (discovered) and soft (user-defined) types. A single hard class can be one of several things:

- The name of an operating system architecture e.g. `ultrix`, `sun4`, etc.
- The unqualified name of a particular host. If your system returns a fully qualified domain name for your host, CFEngine truncates it at the first dot. Note: `www.sales.company.com` and `www.research.company.com` have the same unqualified name – `www`.
- The name of a user-defined group of hosts.
- A day of the week (in the form `Monday, Tuesday, Wednesday, ..`).
- An hour of the day, current time zone (in the form `Hr00, Hr01 ... Hr23`).
- An hour of the day GMT (in the form `GMT_Hr00, GMT_Hr01 ... GMT_Hr23`). This is consistent the world over, in case you need virtual simultaneity of change coordination.
- Minutes in the hour (in the form `Min00, Min17 ... Min45`).
- A five minute interval in the hour (in the form `Min00_05, Min05_10 ... Min55_00`)
- The quarter-hour (in the form `Q1, Q2, Q3, Q4`).
- A day of the month (in the form `Day1, Day2, ... Day31`).
- A month (in the form `January, February, ... December`).
- A year (in the form `Yr1997, Yr2004`).
- A shift in `Night,Morning,Afternoon,Evening`, which fall into six hour blocks starting at 00:00 hours.
- A 'lifecycle index', which is the year number modulo 3 (used in long term resource memory).
- An arbitrary user-defined string.
- The IP address octets of any active interface (in the form `ipv4_192_0_0_1`, `ipv4_192_0_0`, `ipv4_192_0`, `ipv4_192`).

To see all of the classes define on a particular host, run

```
host# cf-promises -v
```

as a privileged user. Note that some of the classes are set only if a trusted link can be established with cfenvd, i.e. if both are running with privilege, and the '`/var/cfengine/state/env_data`' file is secure. More information about classes can be found in connection with `allclasses`.

User-defined or soft classes are defined in bundles. Bundles of type `common` yield classes that are global in scope, whereas in all other bundle types classes are local. Soft classes are

evaluated when the bundle is evaluated. They can be based on test functions or simply from other classes:

```
bundle agent myclasses
{
classes:

"solinus" expression => "linux||solaris";

# List form useful for including functions

"alt_class" or => { "linux", "solaris", fileexists("/etc/fstab") };

"oth_class" and => { fileexists("/etc/shadow"), fileexists("/etc/
passwd") };

reports:

alt_class::

   # This will only report "Boo!" on linux, solaris, or any system
   # on which the file /etc/fstab exists
   "Boo!";
}
```

Classes may be combined with the operators listed here in order from highest to lowest precedence:

'()' The parenthesis group operator.

'!' The NOT operator.

'.' The AND operator.

'&' The AND operator (alternative).

'|' The OR operator.

'||' The OR operator (alternative).

So the following expression would be only true on Mondays or Wednesdays from 2:00pm to 2:59pm on Windows XP systems:

```
(Monday|Wednesday).Hr14.WinXP::
```

Consider the following more advanced example. Promises in bundles of type 'common' are global in scope – all other promises are local to the scope of their bundle.

```
body common control
{
```

```
bundlesequence => { "g","ls_1", "ls_2" };
}

#################################

bundle common g
{
classes:

# The promise "zero" is always satisfied , and is global in scope
"zero" expression => "any";

}

#################################

bundle agent ls_1
{
classes:

# The promise "one" is always satisfied , and is local in scope to ls_1
"one" expression => "any";
}

#################################

bundle agent ls_2
{
classes:

# The promise "two" is always satisfied , and is local in scope to ls_2
"two" expression => "any";

reports:

zero.!one.two::

  # This report @b{will} be generated
  "Success";
}
```

Here we see that class 'zero' is global while classes 'one' and 'two' are local. The report 'Success' result is therefore true because only 'zero' and 'two' are in scope in the 'ls_2' bundle (and the class expression for bundle 'ls_2' requires that both 'zero' and 'two' be true and that 'one' not be true).

5.5 Loops

If you are looking for loops in CFEngine then we need to reprogram you a little, as you are thinking like a programmer! CFEngine is not a programming language that is meant to give you low level control, but rather a set of declarations that embody processes. It's the difference between the gears on a bicycle and the automated transmission in a transporter.

Loops are executed implicitly in CFEngine, but there is no visible mechanism for it – because that would steal attention from the intention of the promises. The way to express them is through lists.

Loops are really a way to iterate a variable over a list. Try the following.

```
body common control

{
bundlesequence  => { "example" };
}

#############################################################

bundle agent example

{
vars:

# This is a list

"component" slist => { "cf-monitord", "cf-serverd", "cf-execd" };

# This is an associative array

"array[cf-monitord]" string => "The monitor";
"array[cf-serverd]" string => "The server";
"array[cf-execd]" string => "The executor, not executionist";

reports:

cfengine_3::

"$(component) is $(array[$(component)])";

}
```

The output looks something like this:

```
/var/cfengine/bin/cf-agent -f ./unit_loops.cf -K

R: cf-monitord is The monitor
```

```
R: cf-serverd is The server
R: cf-execd is The executor, not executionist
```

You see from this that, if we refer to a list variable using the scalar reference operator '`$()`', CFEngine interprets this to mean "please iterate over all values of the list". Thus, we have effectively a 'foreach' loop, without the attendant syntax.

5.6 The main promise types

The following promise types may be used in any bundle:

`vars` A promise to be a variable, representing a value.

`classes` A promise to be a class representing a state of the system.

`reports` A promise to report a message.

These additional promise types may be used only in agent bundles

`commands` A promise to execute a command.

`databases`
A promise to configure a database.

`files` A promise to configure a file, including its existence, attributes and contents.

`interfaces`
A promise to configure a network interface.

`methods` A promise to take on a whole bundle of other promises.

`packages` A promise to install a package.

`storage` A promise to verify attached storage.

These promise types belong to other components:

`access` A promise to grant or deny access to file objects in `cf-serverd`.

`measurements`
A promise to measure or sample data from the system, for monitoring or reporting in `cf-monitord` (CFEngine Nova and above).

`roles` A promise to allow certain users to activate certain classes when executing `cf-agent` remotely, in `cf-serverd`.

`topics` A promise to associate knowledge with a name, and possibly other topics, in `cf-know`.

`occurrences`
A promise to point or refer to a knowledge resource, in `cf-know`.

6 Using CFEngine as a front-end or replacement for cron

6.1 Do I need cron?

The Unix cron command is a useful beast, but a dumb one. One of CFEngine's strengths is its use of classes to identify systems from a single file or set of files. Many administrators think that it would be nice if the cron daemon also worked in this way. One possible way of setting up cron from a global configuration would be to use the CFEngine file editing capability to edit each cron file separately. That would be missing an obvious opportunity however.

A much better way is to use CFEngine's time classes to work like a user interface for cron. This allows you to have a single, central CFEngine file which contains all the cron jobs on your system without losing any of the fine control which cron affords you. All of the usual advantages apply:

- It is easier to keep track of what cron jobs are running on the system when you have everything in one place.
- You can use all of your carefully crafted groups and user-defined classes to identify which host should run which programs.

The central idea behind this scheme is to set up a regular cron job on every system which executes `cf-agent` at frequent intervals. Each time `cf-agent` is started, it evaluates time classes and executes the shell commands defined in its configuration file. In this way we use `cf-agent` as a wrapper for the cron scripts, so that we can use CFEngine's classes to control jobs for multiple hosts. CFEngine's time classes are at least as powerful as `cron`'s time specification possibilities, and they add control over location too. This does not restrict you in any way, See Section 6.5 [Building flexible time classes], page 36. The only price is the overhead of parsing the CFEngine configuration file which is insignificant.

DO I NEED TO USE CRON? No. With CFEngine's `cf-execd` you don't *have* to use cron – CFEngine can schedule itself. Whether you choose to run `cf-execd` in daemon mode, or in wrapper mode is entirely up to you. In the commercial versions of CFEngine, the exec daemon has sophisticated features for reliability. In the Community Edition, you might feel comfortable having something independent watching over CFEngine, especially during binary updates during which live programs can die from faults.

6.2 The single cron job approach

To be more concrete, imagine installing the following '`crontab`' file onto every host on your network:

```
#
# Global Cron file
#
*/5 * * * * /var/cfengine/bin/cf-execd -F
```

6.3 Structuring commands promises

The structure of a promise bundle needs to reflect your policy for running jobs on the system. You need to switch on relevant tasks and switch off unwanted tasks depending on the time of day. This can be done by placing individual actions under classes which restrict the times at which they are executed,

```
promise-type:

      time-based classes::

         Promise
```

For example:

```
bundle agent example
{
commands:

# Exec during the first quarter-hour after noon

  Hr12.Q1::

    "/path/myscript -arg1 -arg2";

# Exec during any second quarter-hour

  Q2::

    "/path/otherscript";

# Exec during the intervals 00:10 through 00:15 and 12:45 through 12:55
# (English says ‘‘and’’, but logic says ‘‘if this interval or that is true’’

  Hr00.Min10_15||Hr12.Min45_55::

    "/path/amongstourscripts";

}
```

If you want to get fancy, you can set parameters for the execution of the script by building a container for it that traps its output and privileges (this applies to root only, since only root has this power to change privilege).

```
bundle agent example
{
commands:

# Exec on the first quarter after noon
```

```
  Hr12.Q1::

    "/path/myscript -arg1 -arg2",

          contain => jail("nobody","true");
}

# ...

body contain jail(owner,devnull)
{
exec_owner => "$(owner)";     # run with this setuid
no_output => "$(devnull)";    # like > /dev/null 2>&1
umask => "77";                # set process umask
}
```

The '`contain`'ment body provides a safe and flexible environment in which to embed scripts.

The time resolution of the classes is limited by how often you execute CFEngine either using cron or `cf-execd`. Five minutes is the recommended scheduling interval.

6.4 Splaying host times

In a network of thousands of computers, many agents could start executing and downloading resources from a server at the same time. For instance, if a thousand cf-agents all suddenly wanted to copy a file from a master source simultaneously this would lead to a big load on the server. We can prevent this from happening by introducing a time delay which is unique for each host and not longer than some given interval; `cf-execd` uses a hashing algorithm to generate a number between zero and a maximum value in minutes which you define, like this:

```
body executor control

{
splaytime => "10"; # Minutes
}
```

If this number is non-zero, `cf-execd` goes to sleep after parsing its configuration file and reading the clock. Every machine's `cf-execd` will go to sleep for a different length of time, which is no longer than the time specified.

A hashing algorithm, based on the fully qualified name of the host, is used to compute a unique time for hosts. The shorter the interval, the more clustered the hosts will be. The longer the interval, the lighter the load on your servers. This 'splaying' of the run times will lighten the load on servers, even if they come from domains not under your control but have a similar cron policy.

6.5 Building flexible time classes

Each time CFEngine is run, it reads the system clock and defines classes based on the time and date (see reference manual).

Time classes based on the precise minute at which cfagent started are unlikely to be directly useful in policy (except in the `cf-execd` schedule). Many things could conspire to delay the precise time at which cfagent were started. The real purpose in being able to detect the precise start time is to define composite classes which refer to arbitrary intervals of time. To do this, we use the `group` or `classes` action to create an alias for a group of time values. Here are some creative examples:

```
classes:  # synonym groups:

"LunchAndTeaBreaks" expression => "!(Saturday|Sunday).(Hr12|Hr10|Hr15)";

"NightShift"        or => { "Hr22", "Hr23", "Night" };

"ConferenceDays"    or => { "Day26", "Day27", "Day29", "Day30" };

"TimeSlices"        or => { "Min01", "Min02", "Min03", "Min10_15"
                            "Min33", "Min34", "Min35" };

"Exception"         not => "Hr12.Min15_20";
```

In the first three examples, the left hand sides of the assignments are effectively the ORed result of the right hand side. Thus if any classes in the braces is defined, the left hand side class will become defined. This provides a flexible and readable way of specifying intervals of time within a program, without having to use '|' and '.' operators everywhere.

6.6 Choosing a scheduling interval

How often should you call your global CFEngine configuration? There are several things to think about:

- How much fine control do you need? Running cron jobs once each hour is usually enough for most tasks, but you might need to exercise finer control for a few special tasks.
- Are you going to verify the entire CFEngine configuration file or just selected promises?

CFEngine has an intelligent locking and timeout policy which should be sufficient to handle hanging shell commands from previous crons so that no overlap can take place.

7 Network services

This chapter describes how you can set up a CFEngine network service to handle remote file distribution and remote execution of CFEngine without having to open your hosts to possible attack using the `rsh` protocols.

7.1 CFEngine network services

By starting the daemon called `cf-serverd`, you can set up a line of communication between hosts, allowing them to exchange files across the network or execute CFEngine remotely on another system. CFEngine network services are built around the following components:

`cf-agent` The configuration engine's only contact with the network is via remote copy requests. It does not and cannot grant any access to a system from the network. It is only able request access to files from the server component.

`cf-serverd`
: A daemon which acts as both a file server and a remote-`cf-agent` executor. This daemon authenticates requests from the network and processes them according to rules specified in the server control body and server bundles containing `access` promises.

`cf-runagent`
: This is a simple initiation program which can be used to run `cf-agent` on a number of remote hosts. It cannot be used to tell `cf-agent` what to do, it can only ask `cf- serverd` on the remote host to run the `cf-agent` with its existing configuration. Privileges can be granted to users to provide a kind of Role Based Access Control (RBAC) to certain parts of the existing policy.

With these components you have everything you need to do effective distribution of resources (provisioning) of systems.

7.2 How services work

7.2.1 Remote file distribution

This section describes how you can set up `cf-serverd` as a remote file server which can result in the distribution of files to client hosts in a secure a reliable manner.

An important difference between CFEngine and other systems has to do with the way files are distributed. CFEngine uses a 'pull' rather than a 'push' model for distributing network files. A majority of systems (probably) for instance, works by forcing an image of the files on one server machine onto all clients. This happens in the manner of an attack – indeed the recipients are often required to open various ports and accept whatever they get. CFEngine will not support this kind of technology as a matter of principle.

With the 'push' approach files get changed when the distributor wishes it and the clients have no choice but to live with the consequences. CFEngine, on the other hand, works by *voluntary cooperation*. Hosts are allowed to remain in control of their defenses and protect themselves against attacks and pushes if they want to.

In fact, CFEngine cannot (by design) force its will onto other hosts, nor can it be forced. In order to distribute it can at best signal all machines and ask them to collect files if they are

willing. In other words, CFEngine simulates a 'push' model by polling each client and running the local CFEngine configuration script giving the host the chance to 'pull' any updated files from the remote server, but leaving it up to the client machine to decide whether or not it wants to update.

Also, in contrast to programs like `rdist` which distribute files over many hosts, CFEngine does not require any general `root` access to a system using the '`.rhosts`' file or the '`/etc/hosts.equiv`' file. It is sufficient to run the daemon as root. You can not run it by adding it to the '`/etc/inetd.conf`' file on your system however. The restricted functionality of the daemon protects your system from attempts to execute general commands as the root user using `rsh`.

To remotely access files on a server you use a `copy_from` attribute in a '`files`' promise:

```
bundle agent example
{
files:

"/var/cfengine/inputs"

  perms => m("600"),
  copy_from => remote_cp("$(master_location)","localhost"),
  depth_search => recurse("inf"),
  action => immediate;

}

# Library template

body copy_from remote_cp(from,server)
{
servers     => { "$(server)" };
source      => "$(from)";
compare     => "mtime";
}
```

Assuming that the `cf-serverd` daemon is running on *server- host*, `cf-agent` will make contact with the daemon and attempt to obtain information about the file. During this process, CFEngine verifies that the system clocks of the two hosts are reasonably synchronized. If they are not, it will not permit remote copying unless `denybadclocks` is false in the server control body.

If `cf-agent` determines that a file needs to be updated from a remote server it begins copying the remote file to a new file on the same filesystem as the destination-file. This file has the suffix '`.cfnew`'.

Only when the file has been successfully collected will `cf- agent` make a copy of the old file, (see `repository` in the Reference manual), and rename the new file into place. This behavior is designed to avoid race-conditions which can occur during network connections and indeed any operations which take some time. If files were simply copied directly to their

new destinations it is conceivable that a network error could interrupt the transfer leaving a corrupted file in place. `cf-agent` places a timeout of a few seconds on network connections to avoid hanging processes.

Normally the daemon sleeps, waiting for connections from the network. Such a connection may be initiated by a request for remote files from a running `cf-agent` program on another host, or it might be initiated by the program `cf-runagent` which simply asks the host running the daemon to run `cf-agent` or `cf-execd` program locally.

7.2.2 Remote execution of `cf-agent`

Occasionally you will want to run `cf-agent` immediately in order to implement a change in configuration as quickly as possible on one or more hosts. It would then be inconvenient to have to log onto every host in order to do this manually.

If your scheduling interval is often enough, this should be unnecessary since CFEngine will already have run by the time you manage to log on – and the parallelism means that an entire network can be altered in minutes without the delay of waiting for centralized control.

But you might want to send a special signal, e.g. run policy with a special class activated on just a few machines. Then a better way is to issue a simple command which contacts the remote host and runs `cf-agent` with role based access control, providing the immediate output on your own screen:

```
host$ cf-runagent -H remote-host -v

output....
```

- You avoid having to log in on a remote host in order to reconfigure it.
- Users other than root can run `cf-agent` to fix any problems with the system, with access granted to individuals and classes.

A potential disadvantage with any such system is that malicious users might be able to run `cf-agent` on remote hosts. The fact that non-root users can execute `cf-agent` is not a problem in itself, after all the most malicious thing they would be able to do would be to check the system configuration and repair any problems. No one can tell `cf-agent` what to do using the `cf-runagent` program, it is only possible to run an existing configuration. But a more serious concern is that malicious users might try to run `cf-agent` repeatedly (so-called 'Denial of Service' attack) so that a system became burdened with running `cf-agent` constantly. To protect against this, the server uses the same `ifelapsed` locks to complement access controls.

7.3 Remote access explained

7.3.1 Server connection

In order to connect to the CFEngine server you need

A public-private key pair.
: To create a key pair, run

```
cf-key
```

An IP (v4 or v6) address.
: You must be online with a configured network address.

A client program
: Both `cf-agent` and `cf-runagent` are clients that can connect to the server.

Permission to connect to the server, and
: The server control body must grant access to your computer and public key by name or IP address, by listing it in one of the lists (see below).

Your public key must be trusted by the server, and you must
: trust the server's public key

  By mutually trusting each others' keys, client and server agree to use that key as a sufficient identifier for the computer.

Permission to access something
: Your host name or IP address must be mentioned in an `access` promise inside a server bundle, made by the file that you are trying to access.

If all of the above criteria are met, connection will be established and data will be transferred between client and server. The client can only send short requests, following the CFEngine protocol. The server can return data in a variety of forms, usually files, but sometimes console output.

7.3.2 Remote access troubleshooting

When setting up `cf-serverd`, you might see the error message

```
Unspecified server refusal
```

This means that `cf-serverd` is unable or is unwilling to authenticate the connection from your client machine. The message is generic: it is deliberately non-specific so that anyone attempting to attack or exploit the service will not be given information which might be useful to them. There is a simple checklist for curing this problem:

1. Make sure that the domain variable is set in the configuration files read by both client and server; alternatively use `skipidentify` and `skipverify` to decouple DNS from the the authentication.
2. Make sure that you have granted access to your client in the server body

```
body server control
{
allowconnects         => { "127.0.0.1" , "::1" ...etc };
allowallconnects      => { "127.0.0.1" , "::1" ...etc };
trustkeysfrom         => { "127.0.0.1" , "::1" ...etc };
}
```

3. Make sure you have created valid keys for the hosts using `cf-key`.
4. If you are using secure copy, make sure that you have created a key file and that you have distributed and installed it to all participating hosts in your cluster.

Always remember that you can run CFEngine in verbose or debugging modes to see how the authentication takes place:

```
cf-agent -v
cf-serverd -v
```

cf-agent reports that access is denied regardless of the nature of the error, to avoid giving away information which might be used by an attacker. To find out the real reason for a denial, use verbose '-v' or even debugging mode '-d2'.

7.3.3 Key exchange

The key exchange model used by CFEngine is based on that used by OpenSSH. It is a peer to peer exchange model, not a central certificate authority model. This means that there are no scalability bottlenecks (at least by design, though you might introduce your own if you go for an overly centralized architecture).

The problem of key distribution is the conundrum of every public key infrastructure. Key exchange is handled automatically by CFEngine and all you need to do is to decide which keys to trust.

When public keys are offered to a server, they could be accepted automatically on trust because no one is available to make a decision about them. This would lead to a race to be the first to submit a key claiming identity.

Even with DNS checks for correct name/IP address correlation (turned off with skipverify), it might be possible to submit a false key to a server.

The server cf-serverd blocks the acceptance of unknown keys by default. In order to accept such a new key, the IP address of the presumed client must be listed in the trustkeysfrom stanza. Once a key has been accepted, it will never be replaced with a new key, thus no more trust is offered or required.

Once you have arranged for the right to connect to the server, you must decide which hosts will have access to which files. This is done with access rules.

```
bundle server access_rules()

{
access:

"/path/file"

  admit   => { "127.0.0.1", "127.0.0.2", "127.0.0.3" },
  deny    => { "192.*" };
}
```

On the client side, i.e. cf-runagent and cf-agent, there are three issues:

1. Choosing which server to connect to.
2. Trusting the identity of any previously unknown servers, i.e. trusting the server's public key to be its and no one else's. (The issues here are the same as for the server.)
3. Choosing whether data transfers should be encrypted (with encrypt).

Because there are two clients for connecting to cf-serverd (cf-agent and cf-runagent), there are also two ways on managing trust of server keys by a client. One is an automated option, setting the option trustkey in a copy_from stanza, e.g.

```
body copy_from example
   {
   # .. other settings ..
   trustkey => "true";
   }
```

Another way is to run `cf-runagent` in interactive mode. When you run `cf-runagent`, unknown server keys are offered to you interactively (as with `ssh`) for you to accept or deny manually:

```
WARNING - You do not have a public key from host ubik.iu.hio.no =
128.39.74.25
        Do you want to accept one on trust? (yes/no)
-->
```

7.3.4 Time windows (races)

Once public keys have been exchanged from client to server and from server to client, the issue of trust is solved according to public key authentication schemes. You only need to worry about trust when one side of a connection has never seen the other side before.

Often you will have a central server and many client satellites. Then the best way to transfer all the keys is to set the `trustkey` flags on server and clients sides to coincide with a time at which you know that `cf-agent` will be run, and when a spoofer is unlikely to be able to interfere.

This is a once-only task, and the chance of an attacker being able to spoof a key-transfer is small. It would require skill and inside-information about the exchange procedure, which would tend to imply that the trust model was already broken.

Another approach would be to run `cf-runagent` against all the hosts in the group from the central server and accept the keys one by one, by hand, though there is little to be gained from this.

Trusting a host for key exchange is unavoidable. There is no clever way to avoid it. Even transferring the files manually by diskette, and examining every serial number of the computers you have, the host has to trust the information you are giving it. It is all based on assertion. You can make it almost impossible for keys to be faked or attacked, but you cannot make it absolutely impossible. Security is about managing reasonable levels of risk, not about magic.

All security is based on a moment of trust at some point in time. Cryptographic key methods only remove the need for a repeat of the trust decision. After the first exchange, trust is no longer needed, because they keys allow identity to be actually verified.

Even if you leave the trust options switched on, you are not blindly trusting the hosts you know about. The only potential insecurity lies in any new keys that you have not thought about. If you use wildcards or IP prefixes in the trust rules, then other hosts might be able to spoof their way in on trust because you have left open a hole for them to exploit. That is why it is recommended to return the system to the default state of zero trust immediately after key transfer, by commenting out the trust options.

It is possible, though somewhat laborious to transfer the keys out of band, by copying '/var/cfengine/ppkeys/localhost.pub' to /var/cfengine/ppkeys/user-aaa.bbb.ccc.mmm (assuming IPv4) on another host. e.g.

```
localhost.pub -> root-128.39.74.71.pub
```

This would be a silly way to transfer keys between nearby hosts that you control yourself, but if transferring to long distance, remote hosts it might be an easier way to manage trust.

7.3.5 Other users than root

CFEngine normally runs as user "root" (except on Windows which does not normally have a root user), i.e. a privileged administrator. If other users are to be granted access to the system, they must also generate a key and go through the same process. In addition, the users must be added to the server configuration file.

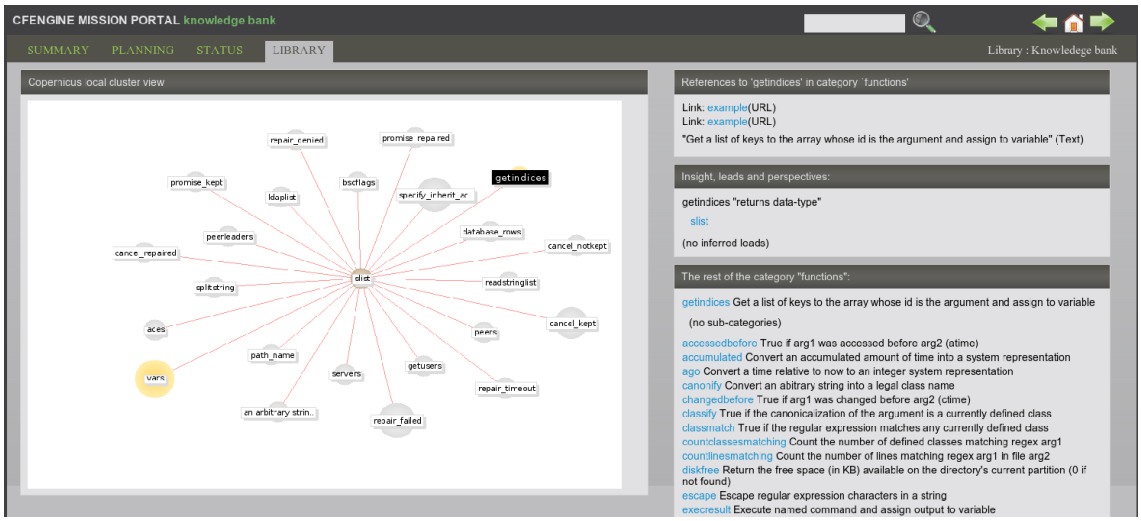
7.3.6 Encryption

CFEngine provides encryption for keeping file contents private during transfer. It is assumed that users will use this judiciously. There is nothing to be gained by encrypting the transfer of public files – overt use of encryption just contributes to global warming, burning unnecessary CPU cycles without offering any security.

The main role for encryption in configuration management is for authentication. CFEngine always uses encrypted for authentication, so none of the encryption settings affect the security of authentication.

8 Knowledge Management

A unique aspect of CFEngine, that is fully developed in the commercial editions of the software, its ability to enable integrated knowledge management as part of an automation process, and to use its configuration technology as a 'semantic' documentation engine.



Knowledge management is the challenge of our times. Organizations frequently waste significant effort re-learning old lessons because they have not been documented and entered into posterity. Now you can alleviate this problem with some simple rules of thumb and even build sophisticated index-databases of documents.

8.1 Promises and Knowledge

The learning curve for configuration management systems has been the brunt of frequent criticism over the years. Users are expected to either confront the informational complexity of systems at a detailed level, or abandon the idea of fine control altogether. This has led either to information overload or over-simplification. The ability to cope with information complexity is therefore fundamental to IT management

CFEngine introduced the *promise model* for configuration in order to flatten out this learning curve. It can lead to simplifications in use, because a lot of the thinking has been done already and is encapsulated into the model. One of its special properties is that it is both a model for system behavior and a model for knowledge representation (this is what declarative languages seek to be, of course). More specifically, it incorporated a subset of the ISO standard for 'Topic Maps', an open technology for semantic indexing of information resources. By bringing together these two technologies (which are highly compatible), we end up with a seamless front-end for sewing together and browsing system information.

Knowledge management is a field of research in its own right, and it covers a multitude of issues both human and technological. Most would agree that knowledge is composed of facts and relationships and that there is a need both for clear definitions and semantic context to interpret knowledge properly; but how do we attach *meaning* to raw information without ambiguity?

Knowledge has quite a lot in common with configuration: what after all is knowledge but a configuration of ideas in our minds, or on some representation medium (paper, silicon etc). It is a coded pattern, preferably one that we can agree on and share with others. Both knowledge and configuration management are about describing patterns. A simple knowledge model can be used to represent a policy or configuration; conversely, a simple model of policy configuration can manufacture a knowledge structure just as it might manufacture a filesystem or a set of services.

8.2 The basics of knowledge

Knowledge only truly begins when we write things down:

- The act of formulating something in writing brings a discipline of thought than often lends clarity to an idea.
- You never confront an idea fully until you try to put it into language.
- Any written record that is kept allows others to read it and pass on the knowledge.

The trouble is that writing is something people don't like to do, and few are very good at. To an engineer, it can feel like a waste of time, especially during a busy day, to break off from the doing to write about the doing. Also, writing requires a spurt of creative thinking and engineers are often more comfortable with manipulating technical patterns and notations than writing fluent linguistic formulations that seem overtly long-winded.

CFEngine tries to bridge this gap by making documentation simple and part of the technical configuration. CFEngine's knowledge agent then uses AI and network science algorithms to construct a readable documentation from these technical annotations. It can do this because a lot of thought has already gone into the meaning of the promise model.

8.3 Annotating promises

The beginning of knowledge is to annotate the technical specifications. Remember that the point of a promise is to convey an *intention*. When writing promises, get into the habit of giving every promise a comment that explains its intention. Also, expect to give special promises *handles*, or helpful labels that can be used to refer to them by in other promise statements. A handle could be something dumb like 'xyz', but you should try to use more meaningful titles to help make references clear.

```
files:

"/var/cfengine/inputs"

     handle => "update_policy",
    comment => "Update the CFEngine input files from the policy server",
      perms => system("600"),
  copy_from => rcp("$(master_location)","$(policy_server)"),
depth_search => recurse("inf"),
file_select => input_files,
     action => immediate;
```

If a promise affects another promise in some way, you can make the affected one promise one of the promisees, like this:

```
access:

"/master/CFEngine/inputs" -> { "update_policy", "other_promisee" },

handle  => "serve_updates",
  admit   => { "217.77.34.*" };
```

Conversely, if a promise might depend on another in some (even indirect) way, document this too.

```
files:

"/var/cfengine/inputs"

      handle => "update_policy",
     comment => "Update the CFEngine input files from the policy
server",
  depends_on => { "serve_updates" },
       perms => system("600"),
   copy_from => rcp("$(master_location)","$(policy_server)"),
depth_search => recurse("inf"),
file_select => input_files,
      action => immediate;
```

This use of annotation is the first level of documentation in CFEngine. The annotations are used internally by CFEngine to provide meaningful error messages with context and to compute dependencies that reveal the existence of process chains. These can be turned into a topic map for browsing the policy relationships is a web browser, using `cf-know`.

> The CFEngine Knowledge Map is only available in commercial editions of the software, where the necessary support to set up and maintain this technology can be provided.

8.4 A promise model of topic maps

CFEngine's model of promises can also be used to promise information and its relevance in different contexts. The Knowledge agent `cf-know` understands three kinds of promise.

`topics:` A topic is merely a string that can be associated with another string. It represents a 'subject to be talked about'. Like other promise types, you can use contexts, which are formed from other topics expressions to limit the scope of the current topic promise.

`things:` Things are a simplified interface to topics, that were introduced to make it easier for users to contribute knowledge about more concrete 'things', or less abstract

ideas. A challenge with knowledge management is the abstract and technical nature of the models one must use to represent it. Things attempt to make that task easier.

`occurrences:`
: An occurrence is a reference to a document or a piece of text that actually represents knowledge content about the topic concerned. Occurrences are generally URLs or strings explaining things or topics.

8.5 What topic maps offer

CFEngine is capable of automating the documentation of a policy, using basic annotations provided above, as a knowledge map. They require very little effort from the user. If you are using the Community Edition of CFEngine, you can develop a topic map, but we do not support the backend technology without a commercial license. In either case, once you become familiar with the use of Topic Maps, you will want to extend your knowledge manually to incorporate things like:

- Local (high level) policy documents
- Related databases, such as CMDBs

So let us spend a while showing how to encode knowledge in topic maps using `cf-know`.

The kind of result you can expect is shown in the pictures below. The example figures show typical pages generated by the knowledge agent `cf-know`. The first of these shows how we use the technology to power the web knowledge base in the commercial CFEngine product.

In this use, all of the data are based on documentation for the CFEngine software, and most of the relationships are manually entered.

For a second example, consider how CFEngine can generate such a knowledge map analysis of its own configuration (self-analysis). The data in the images below describe the CFEngine configuration promises. One such page is generated, for instance, for each policy promise, and pages are generated for reports from different computers etc. You can also create you own 'topic pages' for any local (enterprise) information that you have.

In this example, the promise has been given the promise-handle `update_policy`, and the associations and the lower graph shows how this promise relates to other promises through its documented dependencies (these are documented from the promisees and `depends_on` attributes of other promises.).

The example page shows two figures, one above the other. The upper figure shows the thirty nearest topics (of any kind) that are related to this one. Here the relationships are unspecific. This diagram can reveal pathways to related information that are often unexpected, and illustrates relationships that broaden one's understanding of the place the current promise occupies within the whole.

Although the graphical illustrations are just renderings of semantic associations shown more fully in text, they are useful for visualizing several levels of depth in the associative network. This can be surprisingly useful for brainstorming and reasoning alike. In particular, one can see the other promises that could be affected if we were to make a change to the current promise. Such impact analyses can be crucial to planning change and release management of policy.

A knowledge base is a slightly improved implementation of a Topic Map which is an ISO standard technology. A topic map works like an index that can point to many different kinds of external resources, and may contain simple text and images internally. So you use it to bind together documents of any kind. A CFEngine knowledge base is not a new document format, it is an overlay map that joins ideas and resources together, and displays relationships.

8.6 The nuts and bolts of topic maps

8.6.1 Topic map definitions

Topic maps are really electronic indices, but they form and work like webs. A topic is the technical representation of a 'subject', i.e. anything you might want to discuss, abstract or physical e.g. an item of 'abstract knowledge', which probably has a number of concrete exemplars. It might be a person, a machine, a quality, etc.

Topics can be classified into boxes called *topic-types* so that related things can be collated and unrelated things can be separated, e.g. types allow us to distinguish between `rmdir` the Unix utility and `rmdir` the Unix system-call.

Each typed topic can further point to a number of references or exemplars called *occurrences*. For instance, an occurrence of the topic 'computer' might include books, web documents, database entries, physical manifestations, or any other information. An occurrence is a reference that exemplifies the abstract topic. Occurrence references are like the page numbers in an index.

A book index typically has 'see also' references which point from one topic to another. Topic Maps allow one to define any kind of *association* between topics. Unlike an ordinary index, a topic map has a rich (potentially infinite) variety of cross reference types. For instance,

```
topic_1 ‘‘is a kind of’’ topic_2
topic_1 ‘‘is improved by’’ topic 2
topic_1 ‘‘solves the problem of’’ topic_2
```

The topic map model thus has three levels of containers:

Contexts The box into which we classify a topic to disambiguate different topics with the same name ('in the context of')[1].

Topics/Things
The representation of a subject (an index term).

Occurrence Types
A term that explains how an actual document occurrence relates to the topic is claims to say something about. e.g. (tutorial, manual, or example, definition, photo-album etc).

Occurrences
Specific information resources: these are pointers to the actual documents that we want to read (like page numbers in an index).

Contexts map conveniently into CFEngine classes. Topics map conveniently into promisers. Occurrences also map to promisers of a different type. These three label different levels of

[1] Here, CFEngine differs from the topic map standard in allowing contexts to be overlapping sets, rather than mutually exclusive 'types'. CFEngine is guided by Promise Theory in this respect in order to enable distributed cooperation and the development of a free and emergent ontology.

granularity of meaning. Contexts represent a set of topics that might be relevant, which in turn encompass a set of occurrences of resources that contain actual information about the topics in that context. The primacy of topics in this stems from their ability to form networks by *association*.

The classic approach to information modeling is to build a hierarchical decomposition of non-overlapping objects. Data are manipulated into non-overlapping containers which often prove to be overly restrictive. Topic maps allow us to avoid the kinds of mistakes that have led to monstrosities like the Common Information Model (CIM) with its *thousands* of strictly non-overlapping type categories.

Each topic allows us to effectively 'shine a light' onto the occurrences of information that highlight the concepts pertinent to the topic somehow.

8.7 Example of topics promises

You can use `cf-know` to render a topic map either as text (for command line use) or as HTML (for web rendering). We begin with the text rendering as it requires less infrastructure. You will just need a database.

Try typing in the following knowledge promises:

```
body common control
{
bundlesequence  => { "tm" };
}

#################################################

bundle knowledge tm
{
topics:


"server" comment => "Common name for a computer in a desktop";
"desktop" comment => "Common name for a computer for end users";

programs:: # context of programs

"httpd" comment => "A web service process";
"named" comment => "A name service process";

services::

"WWW" comment => "World Wide Web service",
      association => a("is implemented by",
                       "programs::httpd",
                       "implements");

 # if we don't specify a context, it is "any"

"WWW" association => a("looks up addresses with",
                       "named",
                       "serves addresses to");

occurrences:
```

```
httpd::
   "http://www.apache.org"
     represents => { "website" };

}

#################################################

body association a(f,name,b)

{
forward_relationship => "$(f)";
backward_relationship => "$(b)";
associates => { $(name) };
}
```

The simplified things interface is similar, but uses fixed relations:

```
bundle knowledge company_knowledge
{
things:
 regions::

   "EMEA"     comment => "Europe, The Middle-East and Africa";
   "APAC"     comment => "Asia and the Pacific countries";

 countries::
   "UK"             synonyms => { "Great Britain" },
                 is_located_in => { "EMEA", "Europe" };

   "Netherlands"   synonyms => { "Holland" },
                 is_located_in => { "EMEA", "Europe" };

   "Singapore"     is_located_in => { "APAC", "Asia" };

 locations::
   "London_1"    is_located_in => { "London", "UK" };
   "New_Jersey"  is_located_in => { "USA" };

 networks::

  "192.23.45.0/24"         comment => "Secure network, zone 0. Single octet for corporate o
                   is_connected_to => { "oslo-hub-123" };
```

8.7.1 Analyzing and indexing the policy

CFEngine can analyze the promises you have made, index and cross reference them using the command:

```
# cf-promises -r
```

Normally, the default policy in Nova/Enterprise will perform this command each time the policy is changed.

8.7.2 cf-know

CFEngine's knowledge agent `cf-know` allows you to make promises about knowledge and its inter-relationships. It is not specifically a generic topic map language: rather it provides a powerful configuration language for managing a knowledge base that can be compiled into a topic map.

To build a topic map from a set of knowledge promises in '`knowledge.cf`', you would write:

```
# cf-know -b -f ./knowledge.cf
```

The syntax of this file is hinted at below. The full ISO standard topic map model is too rich to be a useful tool for system knowledge management. However, this is where powerful configuration management can help to simplify the process: encoding a topic map is a complex problem in configuration, which is exactly what CFEngine is for. CFEngine's topic map promises have the following form:

```
bundle knowledge example
{
topics:

topic_type_context::                          # canonical container

"Topic name"                                 # short topic name

      comment => "Use this for a longer description",
  association => a("forward assoc to","Other topic","backward assoc");

  "Other topic";

occurrences:

Topic_name::                                  # Topic

  "http://www.example.org/document.xyz"       # URI to instance

    represents => { "Definition", "Tutorial"}; # sub-types
}
```

The association body templates look like this:

```
body association a(f,name,b)
{
forward_relationship => "$(f)";
backward_relationship => "$(b)";
associates => { $(name) };
}
```

> Promise theory adds a clear structure to the topic map ontology, which is highly beneficial as experience shows that weak conceptual models lead to poor knowledge maps.

8.8 Modeling configuration promises as topic maps

We can model topic maps as promises within CFEngine; the question then remains as to how to use topic maps to model configurations so that CFEngine users can navigate the documented promises using a web browser and be able to see all of the relationships between otherwise isolated and fragmentary rules. This will form the basis of a semantic Configuration Management Database (sCMDB) for the CFEngine software. The key to making these ends meet is to see the configuration of the topic map as a number f promises made in the abstract space of topics and the turning each promise into a meta-promise that models the configuration as a topic with attendant associations. Consider the following CFEngine promise.

```
bundle agent update
{
files:

any::

‘‘/var/cfengine/inputs’’ -> { ‘‘policy_team’’, ’’dependent’’ },

          comment => ‘‘Check policy updates from source’’,
            perms => true,
             mode => 600,
        copy_from => true,
      copy_source => /policy/masterfiles,
          compare => digest,
     depth_search => true,
            depth => inf,
        ifelapsed => 1;

}
```

This system configuration promise can be mapped by CFEngine into a number of other promise proposals intended for the `cf-know` agent. Suppressing some of the details, we have:

```
type_files::

"/var/cfengine/inputs"
    association => a("promise made in bundle","update","bundle
contains promise");
"/var/cfengine/inputs"
    association => a("specifies body type","perms","is specified in");
"/var/cfengine/inputs"
    association => a("specifies body type","mode","is specified in");
"/var/cfengine/inputs"
    association => a("specifies body type","copy_from","is specified
in");

# etc ...
```

```
occurrences:

_var_CFEngine_inputs::

  "promise_output_common.html#promise__var_CFEngine_inputs_update_cf_13"
     represents => { "promise definition" };
```

Note that in this mapping, the actual promise (viewed as a real world entity) is an occurrence of the topic 'promise'; at the same time each promise could be discussed as a different topic allowing meta-modeling of the entity-relation model in the real-world data. Conversely the topics themselves become configuration items or 'promisers' in the promise model. The effect is to create a navigable semantic web for traversing the policy; this documents the structure and intention of the policy using a small ontology of standard concepts and can be extended indefinitely by human domain experts.

9 More...

You will find extensive help, examples and documentation as part of the commercial CFEngine support. Visit the website http://www.cfengine.com for more details.