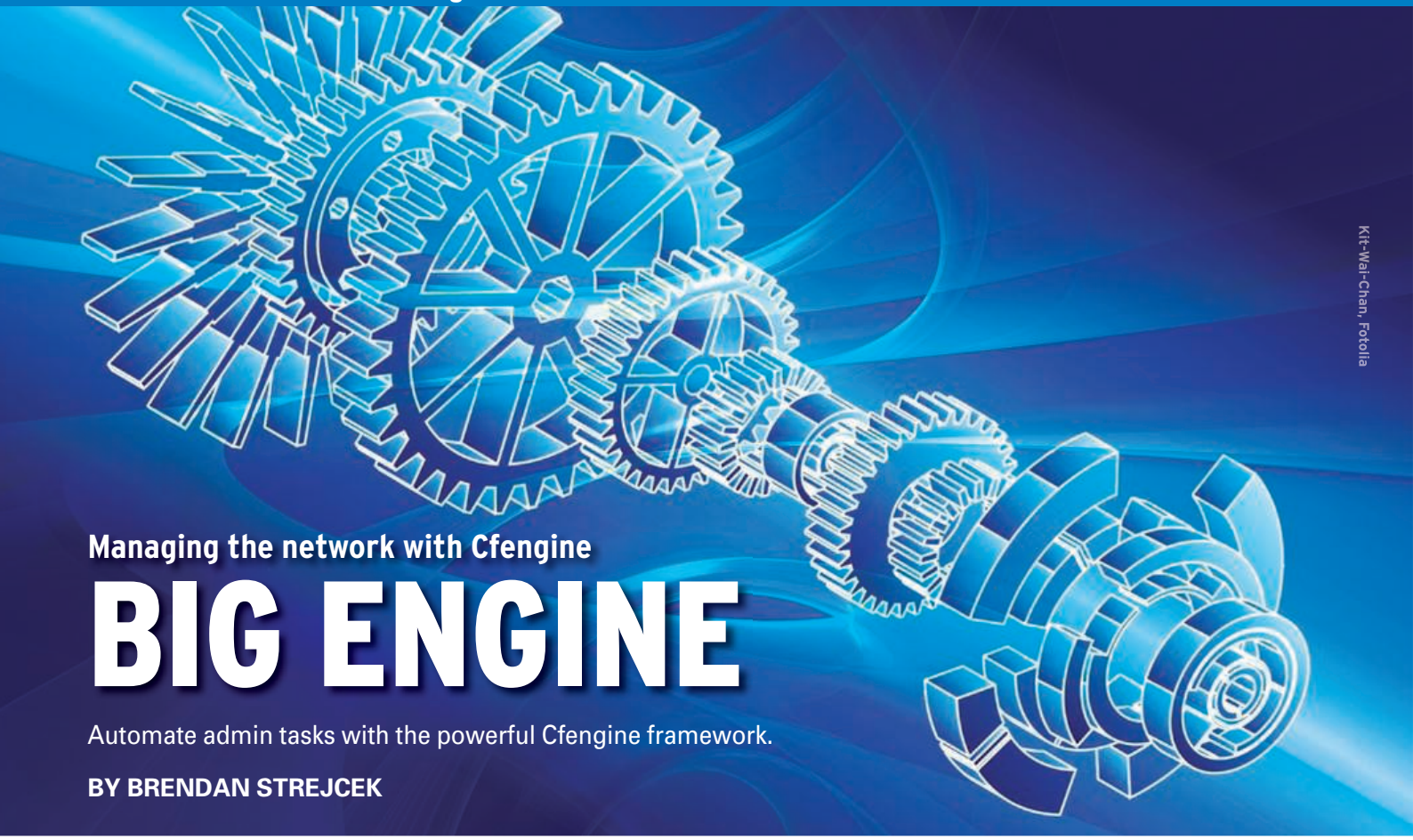

Kit-Wai-Chan, Fotolia

Managing the network with Cfengine

BIG ENGINE

Automate admin tasks with the powerful Cfengine framework.

BY BRENDAN STREJCEK

Cfengine [1] is a flexible framework for automating system administration tasks. With Cfengine, you can manage one machine or a heterogeneous network. The first version of Cfengine was released more than 15 years ago by Mark Burgess, a professor at Oslo University. According to usage estimates, Cfengine has managed more than 1 million computers over the years. Version 3 of the Cfengine framework rolls out some new capabilities and does away with all the old historical layers. The developers have even retooled the language so that all elements are handled in a uniform way.

To show what is possible with Cfengine 3, I introduce various Cfengine components in a running example. To follow along, you need two networked Linux machines that I call *PolicyServer* and *Client*. The end goal is to have the client machine running a fully configured and managed Apache web server, with no manual configuration required, other than installing Cfengine.

The basic model I use will store and distribute all of the policy code centrally from a single server. Cfengine can be used many ways because it is very flexible, but this is a common design, and it serves many sys admins well. *PolicyServer* will hold and make available the central repository of Cfengine code, and the *Client* machine will receive the Apache configuration.

Downloading and Building

First, install Cfengine on both the client and server. Packages are available for many popular Linux distributions, or you can build the tool from source code.

CFengine 3 has very few build-time dependencies and even fewer run-time dependencies (only OpenSSL libcrypto and Berkeley DB libdb). Although not strictly necessary, the Perl-compatible regular expression library (libpcre) also contributes significantly to Cfengine.

If you are building Cfengine from source, first obtain the latest Cfengine 2 and 3 tarballs from the project website [2]. Also, you need Flex, Bison, and Make to compile Cfengine, as well as the static library libcfengine from Cfengine 2. Once you have the dependencies in place, first build Cfengine 2, then Cfengine 3 (you can use the same procedure for both):

```
./configure
make
sudo make install # Or use su
```

Design Principles

Cfengine is built around a number of design principles. In general, the language is descriptive rather than iterative: As much as is possible, you are attempting to describe the "what" of the system rather than the "how." In practice, this approach usually means that Cfengine actions are idempotent; that is, applying the same function twice will result in the same result. This characteristic is important because Cfengine continually monitors the state of your nodes and, depending on how you write your policy, corrects any divergences.

Another principle that Cfengine adheres to is the "pull" architecture, which means that clients request new policy code from a server. This behavior is in contrast to the "push" system, which requires a central node to connect periodically to all clients to configure them. The use of a pull architecture allows you to configure a machine that is down or not yet built because any changes will be picked up automatically when the machine comes onto the network. Cfengine has the facilities to do a push if you really need it, but even these features are built around an underlying pull mechanism. The pull principle also has important implications for the autonomy of the configured node: If the Cfengine server crashes, or if it is unavailable to the client for some reason, the client can continue to use its cached policy until the next time it can connect successfully.

By default, files are installed to */usr/local*, but you can change this by adding the *--prefix=/some/other/path* argument to *configure*, although you need to make sure that the Cfengine build process can find libcfengine. All the binaries are installed in *sbin* under the relevant prefix (by default, */usr/local/sbin*). The next version of Cfengine, 3.0.1, due for release later this year, will not require libcfengine, so this is just an intermediate measure.

Hello, World

With the software safely installed on the server and client, you are ready for a first look at Cfengine in action. A simple "Hello, World" example will demonstrate a working Cfengine 3 program First, run *cf-key* with no arguments. This command creates some dot files in your home directory and generates a keypair (which you won't need right away, but it is necessary for remote copies).

This "Hello, World" program will be written for *cf-agent*, which is the program that does the bulk of the configuration work in Cfengine. *cf-agent* monitors system state and applies corrective action when necessary. Much as *perl* and *sh* binaries are interpreters for Perl and the Bourne shell, respectively, you can think of *cf-agent* as a command interpreter for the Cfengine language. By default, as an unprivileged user, *cf-agent* reads and executes code in *~/.cfagent/input/promises.cf*. So, with your favorite editor, create that file and enter:

```
body common control {
  bundlesequence => { "hello" };
}
bundle agent hello {
reports:
    linux::
        # This is a comment
        "Hello, world.";
}
```

White space does not matter in the Cfengine language, so you can indent this code as you see fit. With no arguments at the shell, go ahead and run it with *cf-agent*. It doesn't matter which of the two machines you run this on because you have installed the software on both.

As you can see, the two main entities present in the code are a *body* named *control* and a *bundle* named *hello*. Bundles are the primary statement aggregation construct in Cfengine (in the same way that a function is the primary construct of C, although bundles are not functions in a mathematical sense). Bodies are groupings of parameters. Both the body and the bundle specify which component of Cfengine they are to be consumed by; in the case of the *control* body, the consumer is *common*, a special keyword meaning the Cfengine suite as a whole, and in the case, of the *hello* bundle, the consumer is *agent*, which refers to the Cfengine binary *cf-agent*.

The name of the bundle, *hello*, is referenced in *bundlesequence*, which is a special directive that tells *cf-agent* what code to execute, and in what order. The special token *reports* is a promise type – one of many kinds of statements that you can make about how you want your system to function.

Bundles are made up of *promises*. In this case, as you can probably guess, *reports* is a way to generate output. The next token, *linux*, followed by a double colon, is a class. Later, I will explain classes in more detail, but for now, just know that code following this class will only execute on a Linux node.

Config Files

Now that *cf-agent* is up and running, the next step is to configure the *cf-serverd* daemon on *PolicyServer* so that the client can download an updated policy. *cf-serverd* functions as a secure file server that provides external access to the *cf-agent* running on a specific node.

On *PolicyServer*, designate a directory as the canonical policy repository. Here, I use */srv/cf-serverd*, but you can select whatever location fits best in your environment. (This should *not* be */var/cfengine*. *PolicyServer* will probably also be a client; that is, the server will update its policy and evaluate it with *cf-agent*.)

Within your central repository, create an *inputs* directory (I am mirroring the contents of the working directory */var/cfengine*, but this is the only subdirectory that I care about for now). In */srv/cf-serverd*, you need to create four files. The first step is to create *cf-serverd.cf* (Listing 1). This file will control which machines can connect to the server and which files they will have access to, and it also will have some *cf-serverd*–specific configuration variables.

Now, create *update.cf*, which will contain code that the client runs to synchronize its local policy to the central policy in the repository (Listing 2).

Listing 2 introduces some Cfengine variables. Unlike past versions, variables are now all dynamic types (before, they were all just strings). Other variable types include *slist* (a list of strings), *real* (a number with decimal precision), and *int* (an integer). Variables are substituted (as in the shell) with *${variablename}*.

Listing 3 shows the file *promises.cf*. The final configuration file is *failsafe.cf*, which simply contains the following:

```
body common control {
    bundlesequence => ➥
    { "update" };
    inputs => { "update.cf" };
}
```

The special *promises.cf* and *failsafe.cf* files are basically just dispatches specifying what other code *cf-agent* should execute. The names for *cf-serverd.cf* and *update.cf* I made up myself (you can call them whatever you want, but I suggest names that are similarly suggestive).

Listing 1: cf-serverd.cf

```
01  body server control {
02      trustkeysfrom => { "192.168.1.62", "192.168.1.61" };
03      allowconnects => { "192.168.1.62", "192.168.1.61" };
04      maxconnections => "10";
05      logallconnections => "true";
06  }
07  bundle server access_rules {
08  access:
09      "/srv/cf-serverd"
10          admit => { "192\.168\..*" };
11  }
```

The hard-coded entry point for *cf-agent* is *promises.cf*. All of the code you want to run needs to be either in this file or referenced by this file. Strictly speaking, *failsafe.cf* is not required, but if *promises.cf* does not parse, *cf-agent* will fall back to *failsafe.cf*, so it is a good idea to make sure that a very simple, known, good *failsafe.cf* is available.

failsafe.cf merely attempts to update the policy files from the server. Because I designed *failsafe.cf* to get only the most recent policy, it also functions as a bootstrap procedure for the Cfengine client. Thus, to configure the client initially, you only need to copy *failsafe.cf* (and any files it references) onto the client.

Get the Keys

Before you can test the system, you need to generate public key cryptography keypairs for each node. As root, run *cf-key* on both the *PolicyServer* and the client. This command will create identities in */var/cfengine/ppkeys*. Because of how *cf-serverd* is configured in *cf-serverd.cf*

```
 trustkeysfrom => ➥
{ "192.168.1.62", ➥
"192.168.1.61" };
```

and *cf-agent* is configured in *update.cf*,

```
trustkey => "true";
```

the behavior of the machines will be to accept the key of a remote node on trust the first time, but from then on only accept clients coming from that same IP address on trust if they can prove these clients have the same key. This stance is rather permissive, but you can tighten up your production systems if you deem it worth the effort. However, then you need to deal with key distribution through an external channel. (One common way to improve this is to distribute the server's public key with the use of your OS install system but to allow the server to accept new clients on trust. Only you can decide what an appropriate level of security is for your site.)

Getting Started

To start, manually, copy *cf-serverd*'s configuration files into place:

```
cd /srv/cf-serverd/inputs
cp promises.cf update.cf ➥
cf-serverd.cf ➥
/var/cfengine/inputs
```

Note that the server is also getting copies of *update.cf* and all the other files; during normal functioning, any changes you make to *cf-serverd.cf* in the central repository will be picked up automatically. With the configuration files in place, start up *cf-serverd*. The following command starts *cf-serverd* verbosely and in the shell foreground. If you leave these options off, *cf-serverd* will silently go into the background.

```
cf-serverd --verbose --no-fork
```

Now, bootstrap the client by copying *failsafe.cf* and *update.cf* manually to */var/cfengine/inputs* on the client (remember, in a production environment, this is something that could be taken care of automatically), then run *cf-agent* to execute the code in *failsafe.cf* from the directory that contains *failsafe.cf*:

```
cf-agent --bootstrap
```

If you switch back to the server console, you should see many messages about what is going on from the server end. If you didn't configure your access control correctly, diagnostics explain why the connection or copy was denied. Once you have verified that the network copy was successful, you can kill the foreground *cf-serverd* process (Ctrl + C) and start it up as a daemon by running it with no arguments.

Scheduler

The last bit of Cfengine infrastructure is the periodic scheduler. *cf-execd* is a scheduler daemon similar to cron. Perhaps you wonder why Cfengine doesn't just use cron. In fact, many people run Cfengine out of cron as well for an added level of reliability, and they configure *cf-agent* to restart either *crond* or *cf-execd* if necessary. The use of *cf-execd* has a number of benefits, though, including the power to control the execution schedule within the central Cfengine policy, as well as the ability to format and send email reports about any actions. If you do decide to run *cf-agent* with cron as well, I recommend having cron execute *cf-agent* via the foreground version of *cf-execd*; that way, you will get the same email settings in both systems, and *cf-execd* will log any output in */var/cfengine/outputs*. In this case, however, I assume you are only running *cf-agent* out of *cf-execd* and not cron.

First, create a new file that controls the functionality of *cf-execd* (*cf-execd.cf*) and add it to the inputs list (Listing 4). Listing 4 states that *cf-execd* will run

Listing 2: update.cf

```
01  body copy_from
      remote(server, path) {
02    servers => {
        "${server}" };
03    encrypt => "true";
04    trustkey => "true";
05    source => "${path}";
06    compare => "digest";
07    preserve => "true";
        # Preserve permissions
08    verify => "true";
09    purge => "true";
10  }
11  body depth_search recurse
     {
12    depth => "inf";
13  }
14  bundle agent update {
15  vars:
16    any::
17    "cfserverd" string =>
        "192.168.1.61";
18    "policyfiles" string =>
        "/srv/cf-serverd";
19   "server_inputs" string =>
       "${policyfiles}/inputs";
20   "client_inputs" string =>
       "${sys.workdir}/inputs";
21  files:
22    any::
23    "${client_inputs}"
24    copy_from =>
        remote("${cfserverd}",
        "${server_inputs}"),
25    depth_search =>
        recurse;
26  }
```

twice per hour (that is the "schedule" line, and the two members of the list are called Cfengine time classes) and that it will send mail to *root@example.com* with *92.168.1.61* as a relay. If you have a usable email address and relay, I recommend using them to get a feel for how the whole system produces feedback for the admin.

The item most in need of explanation is *splaytime*. If a *splaytime* is set, *cf-execd* effectively waits a pseudo-random number of minutes before attempting to connect to the server, with the *splaytime* number as a ceiling. So, in Listing 4, *cf-execd* waits up to 1 minute. The point is to avoid resource contention.

In this case, I have set it to the artificially low value of *1* so that the user will not need to wait long to see activity from *cf-execd*. In a production environment, it would probably be better to set this to something on the order of 15 or 20 minutes for the schedule given in Listing 4.

Remember to add the *executor* bundle that you just created to a *bundlesequence* in *promises.cf*. Now, go back to the client and run *cf-agent* again. This should update the policy from the server and execute it.

Afterwards, check a process listing and you should see that *cf-execd* was started. From the client's point of view, the process is now truly "hands-off": Any modifications you make to the central policy repository will be picked up automatically. Once the scheduled time comes, *cf-execd* will wake up, run *cf-agent*, and deposit any output in */var/cfengine/outputs*.

Maintaining a Service

Suppose I want to use Cfengine to install and configure Apache httpd. In fact, I will even build httpd from source so the solution will be portable across many distributions and platforms. In a production environment, I would hesitate to have servers compile their own software. If I truly needed to build from source, I would most likely build a custom package and then distribute that. However, the use of *cf-agent* to build from source directly offers a nice (cross-platform) way to display some of the available features.

First, download the Apache source into the Cfengine repository [3].

Rather than configuring the client to download the source from the Internet, it is better to cache the source code locally, so you are not dependent on external resources. Just put the tarball in */srv/cf-serverd/inputs* on *PolicyServer* (in a subdirectory for good organization), then let the update bundle take care of distributing it.

Create a new file to store all the httpd-related code – say, *web_server.cf*. This file needs to be added to inputs in *promises.cf*, and any bundles contained within it to *bundlesequence*. The first step is to create a bundle with some variables that can be re-used by other bundles. A bundle of type *common* can be consumed by any Cfengine component and need not be listed in *bundlesequence*. Each bundle has its own scope, and variables from a foreign bundle can be accessed with the interpolation form *${bundlename.variable}*. So, the code in Listing 5 allows other bundles to make use of, for example, *${httpd.conf}*, which will evaluate to the full path.

Particular promises, such as commands, files, or reports, often have parameters that determine the nature of the promise. The appropriate key/value pairs follow the promise. For example, consider the following promise:

```
processes:
    any::
        "cf-execd"
            restart_class => ➥
            "start_cfexecd";
```

This promise has a parameter called *restart_class* that takes a string value for its right-hand side. (In this case, that string will become a defined class if no *cf-execd* processes are running.) Some parameters take external bodies for their right-hand side. The use of an external body allows multiple key/value pairs and further parameterization, which allows reuse. To make the concept concrete, consider the example that I will soon use to compile Apache. The following body, which takes one argument, allows me to run commands in a particu-

Listing 3: promises.cf

```
01  body common control {
02    bundlesequence => {
        "update" };
03    inputs => { "update.cf",
        "cf-serverd.cf" };
04  }
05  # Some arbitrary harmless
    actions that will generate some
    output
06  bundle agent hello {
07  commands:
08    any::
09     "/bin/date";
10     reports:
11      linux::
12       "Hello, world.";
13  }
```

Listing 4: cf-execd.cf

```
01  body executor control {
02      splaytime => "1";
03      mailto => "root@example.com";
04      smtpserver => "192.168.1.61";
05      mailmaxlines => "1000";
06      schedule => { "Min00_05", "Min30_35" };
07      executorfacility => "LOG_DAEMON";
08  }
09  bundle agent executor {
10  processes:
11      any::
12          "cf-execd"
13              restart_class => "start_cfexecd";
14  commands:
15      start_cfexecd::
16          "/usr/local/sbin/cf-execd";
17  }
```

lar directory and without a shell:

```
body contain cd(dir) {
    useshell => "false";
    chdir => "${dir}";
}
```

Such bodies can be stored in any Cfengine input file, but because they are often general and can be reused by many promises, it makes sense to keep them in their own file, which I will call *library.cf*. If you have not already done so, put this *cd* body in *library.cf* and add it to the *bundlesequence* in *promises.cf*. Remember, when changing such an external body later, you might be affecting numerous active promises, so it makes sense to treat them with the care afforded to any shared resource.

In Cfengine, a class is a boolean condition meant to represent some aspect of the system state, be that state an operating system or the time of day. Many classes are defined automatically by *cf-agent*, and you can define others from the return values of programs and by other means. Any promises following a class expression (strings ending with ::) are only enforced when the class is true. For example, read

```
bundle agent a ➥
{ reports: linux:: "asdf"; }
```

as "print *asdf* if the class *linux* is defined." As it happens, *cf-agent* automatically defines the class *linux* on Linux nodes. The special class *any* has been used several times already; this class is always true. It is often used, even when not strictly necessary, to maintain correct indentation. By running *cf-agent -pv* (this will not execute policy code, so it is always safe), you can see all the automatically defined classes. On one of my test nodes, some of the automatically defined classes are: *64_bit*, *Friday*, *debian_4*, and *xen*.

Apache Example

Listing 6 shows the bundle that will unpack, compile, and install Apache. On most systems, the special predefined variable *sys.workdir* will resolve to */var/cfengine*, which essentially says: Test to see whether the software is installed by checking for a particular file (more precise heuristics could be devised); if not, build the program with the standard *untar*, *configure*, *make*, and *make install* procedure as usual.

Many server applications come with configuration files that must be in place before a complete service is deployed. In this case, I will configure Apache to allow *server-info* and *server-status* requests. This requires editing two different configuration files. Cfengine 3 includes four types of promises that reside in special external *edit_lines* bundles – *delete_lines*, *replace_patterns*, *field_edits*, and *insert_lines* – and support additional parameters.

With these promises, you can set configuration variables, comment out key lines, and maintain configuration files. Before you can use *edit_lines* in a "files" promise, you need to create some *edit_lines* bundles. Think of these *edit_lines* bundles as custom-made *editfiles* functions; they are usually general enough to re-use over many components. Two that I will make use of are *DeleteLinesContaining* and *ReplaceAll*. If you are following the file organization I have been using so far, it makes sense to put these in the *library.cf* file with other shared bodies (Listing 7). As you can see, they have pretty much the same structure as other bundles, and they can be parameterized as well.

In addition, I need a way to define a class that, if I edit any files, lets me trigger a service restart later:

```
body classes ➥
```

Listing 5: The common Bundle

```
01  bundle common httpd {
02  vars:
03      any::
04          "version" string => "httpd-2.2.10";
05          "prefix" string => "/opt/httpd/${version}";
06          "server" string => "${prefix}/bin/httpd";
07          "apachectl" string => "${prefix}/bin/apachectl";
08          "conf" string => "${prefix}/conf/httpd.conf";
09  }
```

Listing 6: Setting Up Apache

```
01  bundle agent install_web_server {
02  vars:
03    any::
04      "source" string =>
05        "${sys.workdir}/inputs/support_files/
            ${httpd.version}.tar.gz";
06
07      # Will get automatically cleaned up by the
            update purge
08      "compiledir" string => "${sys.workdir}/
                    inputs/${httpd.version}";
09  classes:
10    "web_server_installed" expression =>
       fileexists("${httpd.server}");
11  commands:
12   !web_server_installed::
13    "/bin/tar xzf ${source} -C
          ${sys.workdir}/inputs";
14    "/bin/sh configure --prefix=
          ${httpd.prefix} --enable-modules=all"
15    contain => cd("${compiledir}");
16    "/usr/bin/make"
17    contain => cd("${compiledir}");
18    "/usr/bin/make install"
19    contain => cd("${compiledir}");
20  }
```

Listing 7: library.cf

```
01  bundle edit_line DeleteLinesContaining(pattern) {
02  delete_lines:
03      ".*${pattern}.*";
04  }
05  body replace_with ReplaceValue(value) {
06      replace_value => "${value}";
07      occurrences => "all";
08  }
09  bundle edit_line ReplaceAll(from,to) {
10  replace_patterns:
11      "${from}"
12          replace_with => ReplaceValue(${to});
13  }
```

Listing 8: httpd.conf

```
01  bundle agent configure_web_server {
02  classes:
03      "web_server_installed" expression =>
            fileexists("${httpd.server}");
04  vars:
05      "info" string => "Include conf/extra/httpd-info.conf";
06  files:
07      web_server_installed::
08         # Uncomment httpd-info.conf line
09         "/tmp/httpd.conf"
10            edit_line => ReplaceAll("^#${info}.*", "${info}"),
11            classes => DefineIfChanged("restart_httpd");
12         # Remove access control from httpd-info.conf
13         "/tmp/httpd-info.conf"
14            edit_line => DeleteLinesContaining("(Allow|Order|Deny)"),
15            classes => DefineIfChanged("restart_httpd");
16  commands:
17      restart_httpd::
18         "${httpd.apachectl} graceful";
19  }
```

Listing 9: Watchdog

```
01  bundle agent monitor_web_server {
02  classes:
03     "web_server_installed" expression =>
           fileexists("${httpd.server}");
04  processes:
05     web_server_installed::
06        # Define a class if httpd is not running so that we can start it
07        "httpd"
08            restart_class => "start_httpd";
09  commands:
10     start_httpd::
11        "${httpd.apachectl} start";
12  }
```

```
DefineIfChanged(class) {
       promise_repaired => ➥
       { "${class}" };
   }
```

Once I have all these components in place, I can tell *cf-agent* to use them to edit the config files; in this case, I need to uncomment the *httpd-info* line in *httpd.conf* and remove the access control from *httpd-info.conf* (Listing 8).

Watchdog

If you need a way to keep an eye out for the web server process – to restart it if it is not running – create another bundle, or simply add the promises in Listing 9.

The code in Listing 9 wraps the process detection with a class so I am sure the web server is running on nodes that have a web server installed.

For even greater reliability, you might want to create a functional test – that is, a test that queries the service. In this case, you need to fetch some data from port 80 and make sure it is the data you expect.

Conclusion

Now that your Cfengine framework is configured, here are a few ideas for continued improvements:

- Centralize periodically executed jobs
- Integrate the monitoring and deployment systems by having *cf-agent* automatically configure monitors
- Integrate your backup system with your deployment system
- Make sure all of your nodes are configured to log centrally

The more functionality you bring within Cfengine's realm, the easier it will be to bring new services online and to recover from problems such as hardware failures or security compromises. Because you can code all the rules on how to create a node of type *X* in a machine-executable language, all you need to do is prepare a fresh base OS install, then install Cfengine and let it rebuild your replacement node for you. ■

INFO

[1] Cfengine: *http://www.cfengine.org*

[2] Cfengine source code: *http://www.cfengine.org/downloads/*

[3] Apache tarball: *http://httpd.apache.org/download.cgi*