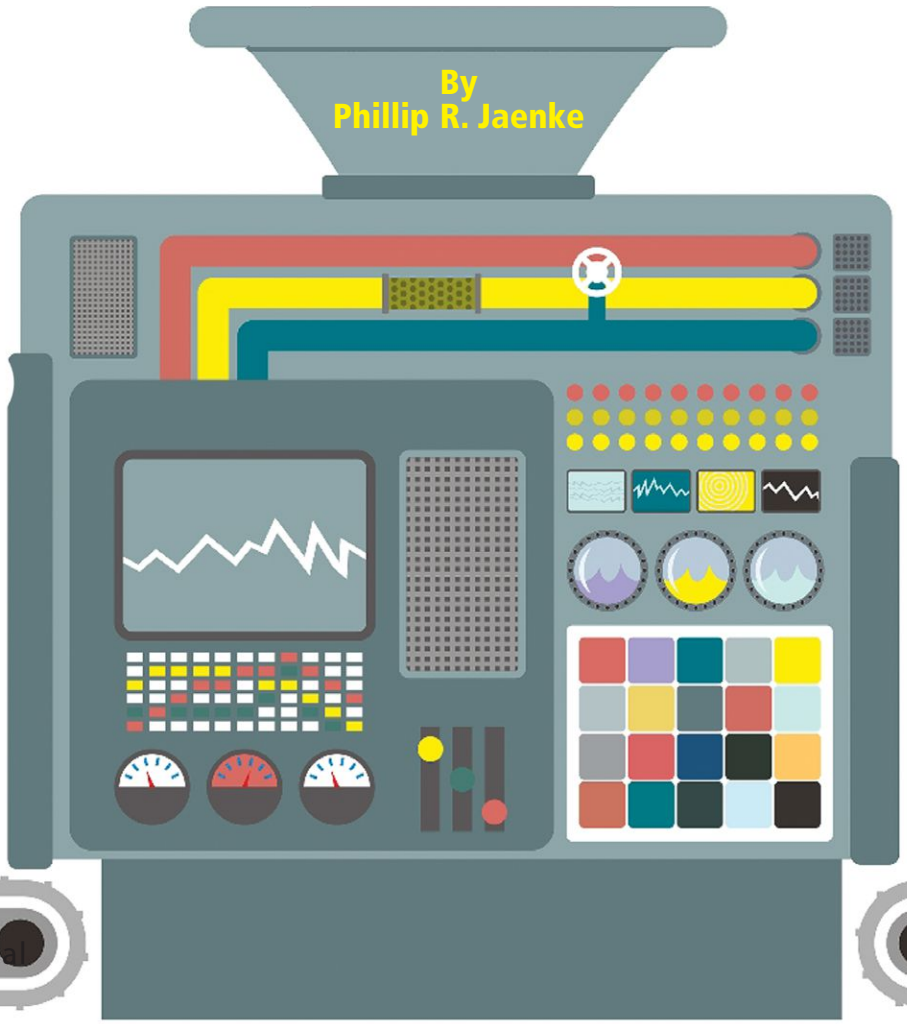
SEE TEXT ONLY

Yes, CFEngine

CAN MANAGE THAT

Configuration management is, as you might have guessed, currently a hot topic in IT. With the advent of people spinning up VMware guests by the dozen and another 50 instances in AWS for good measure, it's become more of a necessity than ever before. Once a setup with 200 environments on 2 dozen hosts (because jails really are that great) was considered a large environment. It was also something a pair of skilled system administrators could easily maintain, because they built every one of those environments. Sure, you had lots of one-offs, but you could easily track them with a text file here and a post-it note there. (Do not disable password logins on clx9oa because they access that from the AS/400.)

By
Phillip R. Jaenke



These days, "large" environments start at about 20,000 hosts and only go up from there. As you might imagine, no amount of skill will ever make up for that kind of volume. And these environments often go back more than a decade. It is guaranteed that at some point you will poke your head into a troublesome host only to find out that it's a Sun E420R running Solaris 2.5.1, despite the company having standardized on FreeBSD and Windows more than a decade back. Also, it's the one singular system storing your entire login database—for all the employees and customers.

These problems are certainly not new or unknown. The scope and scale have changed tremendously over the years, but the notion that setups like this exist and cause untold numbers of WTF-o'clock calls is almost as old as a moth in the relay. A gentleman doing a post-doctoral fellowship at Oslo University College realized that this was a real problem, so he became a monk instead—wait, no, that's just what a lot of us wished we could do. Instead, Mark Burgess decided to come up with a solution, and, in 1993, released the very first iteration of CFEngine.

Next to shell scripts and post-it notes on the monitor (and maybe some obscure offering from IBM), CFEngine is the oldest and most mature of all configuration management solutions out there. And not by a little, either. The next oldest you'll find discussed in this issue is Puppet, the initial release of which occurred 12 years *after* CFEngine. And immediately, those of us who have been down the single glass of pain and "mature solution" road cringe at the horror of 1990's legacy C.

While it's true that CFEngine is written in C, it's for portability and performance reasons. Other configuration managers are primarily or exclusively based around their own code and the general concept of configuration management. CFEngine's longevity has come about because it is based around two key concepts: convergence and Promise Theory. Rather than say "things are bad, make them all the same in an easy fashion," Mark Burgess sought to understand the ad hoc choices that had been made initially and a method for understanding those choices.

You can read about all that on Wikipedia of course. But that is the nuts and bolts of why CFEngine has given users a consistent and stable set of expectations and behavioral concepts for more than two decades and two complete rewrites. Because it is an implementation of concept and theory, the details of how it does so are far less important.

A Useful Hello World in CFEngine

Because CFEngine uses bidirectional communications (agent and server model) and is specifically targeted at heterogeneous environments, our example here will show you how to use CFEngine's Domain Specific Language to write a 'Hello, World' promise that tells you a bit about the system as well.

To start, you need to understand how the process works. First, you use your policy files to define the desired state. The agent then ensures the state (every 5 minutes). Then the server may optionally verify the desired state, if you're using CFEngine Enterprise. ObPlug: Enterprise is completely free for up to 25 hosts, but unfortunately requires your policy hub be Linux.

The Promise

```
## This is a CFEngine Hello World Promise
bundle agent hello_world
{
    reports:
          any::
                "Hello World";
}
```

This promise establishes a bundle called 'hello_world' that will report on any class of hosts. Now we're going to do something a bit more elaborate leveraging CFEngine's built-in classes, called hard classes: (next page)

```
## Fancy CFEngine Hello World Promise
bundle agent hello_world_freebsd
{
      reports:
            any::
                  "Hello World";
            freebsd::
                  "Thanks for reading FreeBSD Journal!";
            windows::
                  "I think I got lost.";

}
```

This policy will do two additional steps. If the system is a member of the FreeBSD hard class, it offers a different greeting from a windows member, while all others offer the default greeting. Let's see this policy in action on two different hosts now.

```
root@freebsd11 # /var/cfengine/bin/cf-agent --no-lock --file
./hello_world.cf  --bundlesequence hello_world_freebsd
R: Hello World
R: Thanks for reading FreeBSD Journal!

[root@centos7] # /var/cfengine/bin/cf-agent --no-lock --file
./hello_world.cf --bundlesequence hello_world_freebsd
R: Hello World

root@freebsd11# /usr/local/sbin/cf-agent --no-lock -D windows --file
./hello_world.cf  --bundlesequence hello_world_freebsd
R: Hello World
R: Thanks for reading FreeBSD Journal!
R: I think I got lost.
```

As you can see, CFEngine's class mechanism is actually quite easy to understand and leverage. This enables you, as the person in charge of preventing chaos, an easy and reliable means to differentiate your hosts within your policies. Because you can also use promises to define soft classes by virtually any means you like, this gives you a combination of flexibility and consistency that many other configuration management solutions often lack or make difficult to implement.

But as with most Hello World examples, this barely scratches the surface of the capabilities available.

Promises and Policies and Bundles, Oh Why?!

When you work with CFEngine, you might be editing files, but those files are inextricably linked to the core concept of Promise Theory. The gist of Promise Theory is that autonomous actors will achieve voluntary cooperation, which sounds about as far from actually doing configuration management as you can possibly get. Words like "voluntary" and "autonomous" and "cooperation" sound weird in this context.

So, to start, let's break down what Promise Theory really is. As you might have guessed, Mark Burgess proposed it as a method of solving problems in obligation-based schemes for policy-based management (or more accurately, to explain CFEngine's operational model). Rather than have configuration directives dictated from a single server or single group of authoritative servers, every member in the group has autonomy of control. That is, they can't be forced into specific behaviors by deterministic commands. Instead, the agent can only make promises about its own behavior and no other system's behavior.

CFEngine agents are not slaving themselves to a single central server, but rather, an agent promises to retrieve data and act on instructions from another server. A good analogy would be a typical everyday occurrence on the job. Your intern promises he will fix the problem on all 50 systems if you tell him how to do it. You give the intern a list of basic steps—edit these files, restart these daemons. The intern then performs the steps exactly as you wrote them. Because he is an intern, you then verify that the results are what was desired. That is an (admittedly very simplified) example of Promise Theory in action.

You're not cooperating due to any explicit obligation or requirement aside from wanting the system in a specific state; an intern does not make any promises about any actions besides his own; you do not sit there watching over the intern's shoulder or enter the commands for him—you simply tell the intern what you want him to do; the intern reports back only the specific information he directly knows; you then validate the instructions had the desired result. I know the whole "intern volunteering and actually understanding what he did" part is probably farfetched, but you get the point.

Now imagine an army of helpful, cooperative, knowledgeable interns. That's pretty much the CFEngine model. Thousands of interns asking

root, "Hey, what do you want me to do? Okay, did it, here's the result."

Having an understanding of that, it becomes much easier to understand how promise, policy, and bundles fit together in CFEngine. In the simplest terms, a policy is a collection of promises. If I have promises called "abc" and "123", I can then combine them into a policy called "321cba" that incorporates both of those promises. If you apply a policy with no promises, nothing will happen.

Our Hello World above is an example of a promise. To incorporate that promise into a policy, I would use a body statement with a bundlesequence like this:

```
body common control
{
    # These are the promises we want the agent to act on
    bundlesequence => { "hello_world_freebsd" };
}

bundle agent hello_world_freebsd
{
...
}
```

A bundlesequence can be thought of as a policy. CFEngine will execute the promises contained within the bundlesequence statement in order. I can then use those promises to control the behavior of other promises. For example, I might use a promise that finds out if the system's hostname begins with the letters qa, and then installs a different sudoers file if that promise is true. I can also place the hostname check entirely within the sudoers promise itself.

But what is a bundle, and how does it fit into things? Especially since we just said promises go into policies! Well, a bundle is a means of collecting similar or related promises into a single promise. For example, my policy may promise that any:: systems receive the bundle "standard_users". Within that bundle, I would then define appropriate promises to cover each user and OS.

Bundles, in other words, provide a means to make many promises based off a single policy decision. The caveat of a bundle is that each promise within the bundle cannot be treated as a separate promise, which is to say that you cannot reuse any of those promises outside of the bundle, unless you make it a separate stand-alone promise. You also cannot use it if all the promises need to be checked before another bundle.

This makes bundles very useful for broad stroke configuration items. But a bundle promise can do everything a stand-alone promise can do as well. So, those broad strokes can actually be extremely system specific—such as setting the correct IP address, registering it in Active Directory, or even joining it to a cluster. It all depends on how you write the promises themselves.

So How Do I Stand It Up?

As we just mentioned (and you probably noticed), CFEngine is an agent-based architecture. Agents will check in with a server—this server is called your Policy Server. The Policy Server is the central repository for not only all your policy files, but any other file you want to distribute. Additionally, with Enterprise, you can use it to distribute the CFEngine binaries or packages to all your agent systems—automatically selected by OS, version, and endianness thanks to hard classes!

Except for Enterprise (called Nova Hub), which has additional interfaces and reporting tools, literally any system that can run CFEngine can be a Policy Server. There are only two basic requirements. One, it must run a version of CFEngine that is compatible with your clients. Two, it must be able to hold any files you want to distribute using CFEngine itself. (Obviously if you prefer to use HTTP/HTTPS distribution or NFS for large files, you can cheerfully ignore this.)

Setting up your Policy Server is similarly simple in the grand scheme of things. First, you will need to build and install CFEngine from ports. Then, build and install the corresponding CFEngine masterfiles from ports. So, for 3.7 you would use `sysutils/cfengine37` and `sysutils/cfengine-masterfiles37`. Note that because of some quirks, CFEngine cannot actually run from /usr/local so you will need to first copy or symlink the binaries from `/usr/local/sbin/` to `/var/cfengine/bin/`. Finally, enable CFEngine in your rc.conf and run the command `/var/cfengine/bin/cf-agent --bootstrap --policy-server 127.0.0.1`

You have now successfully built your Policy Server!

Really, that's it. You're now ready to start writing your policies and connecting clients—which is as simple as changing the IP address of the policy-server argument you just ran above. CFEngine's communications are fully encrypted and automatically handle all the hairy work of creating and maintaining secure keys for you.

One of the things that often catches people off guard about CFEngine is that clients are survivable.

Let's say you need to take your Policy Server down to replace a failed DIMM. (And not because Jr. SysAdmin Jimmy ran a random shell script he downloaded off the Internet as root. By the way, there is now an opening for a Jr. SysAdmin.) But your environment keeps trucking and keeps enforcing.

Because—as part of Promise Theory—CFEngine agents are also autonomous nodes. Once they have successfully bootstrapped, they will continue to enforce the policy they have. If the server should happen to be down, they'll shrug, say, "Well I don't have a new policy," and keep enforcing what they have. They'll try to reconnect opportunistically of course, but they won't stop working.

As an example, let's say your policy says that root must have a password of r34lly$ecur3 and the whole network goes down. Sensing opportunity, Bob the Developer (we ALL have a Bob) manages to boot from a CD and set the root password to b0b0wnz! on a client because he doesn't like not being root. When that system reboots, it will promptly record that somebody screwed with the password, reset the password, and tell the Policy Server, "Hey, somebody tried to change this password," as soon as it's able. Even if somebody yanks the Ethernet cable and sets the switch on fire.

So About Those Policies...

There's a reason I keep skipping over policies in detail, and that's because they truly are flexible in CFEngine. Because every possible aspect of a system can be expressed in a policy using domain specific-language (DSL) (and nearly any state as well), there is basically no limit to what you can do in your policies.

Determining what you want to implement through policies in your own network is something only you can decide (and hopefully without the accountants looking over your shoulder). That flexibility, of course, also makes it virtually impossible to really explain or describe what you can do. For every example, there are more than a dozen other ways to do it.

Maybe you want to assign your FreeBSD systems classes based on a regexp against their hostname using a policy—you can do that! Maybe you want to do it statically instead—also possible! The only real limit to what you can do with CFEngine is your imagination (and the time you want to spend writing policies). Simplicity is usually the best answer, not because you can't do it, but because you can easily spend the rest of your life doing nothing but banging away on one policy.

Because of this power, most folks opt to also implement one of the many excellent third-party promise libraries to ease their workload. The most popular ones are Neil H. Watson's Evolve Thinking library, Normation's ncf library, and, of course, the CFEngine Community Open Promise-Body Library (aka cfengine_stdlib.cf) that you already installed as part of masterfiles.

All of these libraries are implemented completely in CFEngine's DSL, which makes them OS agnostic. You use the same promise libraries for every system regardless of release and OS. Naturally, there are the usual caveats of CFEngine version compatibility and some functions within those libraries being OS-specific. But because they are written in DSL, they will run on nearly every OS with no changes, and will not run on OSes they do not apply to.

So instead of wondering whether you can do something in policy (you can!), the decision-making process becomes about what should be implemented in policy. Complete and utter madness, right? Making decisions based on what suits your environment instead of what the vendor of the week isn't currently apologizing for not actually delivering? What next, accommodating all those special snowflakes without breaking everything? Oh. Right. I already mentioned you can do that too.

So, let's talk about a practical example: my environment. I run FreeBSD, AIX, Linux, VMware, HyperV, and Windows. It's obviously a complex environment to begin with, made more complex by using NFSv4—so Kerberos is mandatory. I also use automount for home directories, Jenkins, an actual poudriere cluster (no, really!), <Insert Mandatory Cloud Buzzword Here>, and Juniper because—you know why Juniper.

When I stand up a new FreeBSD system, I use a template in VMware that already has cfengine37 installed. All I do is add the MAC address for vnx0 to my policy that associates it to a hostname. I bootstrap the agent with one command, and CFEngine does all of this for me:

- Kills dhclient and installs the correct /etc/resolv.conf
- Configures the IPv4 address and defaultrouter for vnx0 using DNS
- Configures the IPv6 address and default router for vnx0 using DNS
- Installs the correct pkg repository configuration and updates any packages from base
- Installs and configures the latest version of the support packages I need—krb5, nss-pam-ldapd-sasl, kstart, etcetera—that don't belong in the template

• Installs the correct /etc/krb5.conf and retrieves the machine's keytab using DNS
• Sets the root password to whatever it is this month (or week, if I forgot it again)
• Installs the correct autofs script and enables autofs in /etc/rc.conf
• Sends me a report telling me that the system successfully bootstrapped and detailing the configuration

What happens after that? That's my baseline policy, so it keeps doing that. If I update the root password promise on the policy server, it will change the root password. If I change the IP address in DNS, it will update rc.conf for me. The real magic is that every OS gets the exact same treatment, with the promise adjusted to fit based on the class returned by the agent. (OK, also the real magic is CFEngine on JunOS, but that's between just you and me.)

Because you can combine promises, bundles, and policy into virtually anything, there is really no limit to what you can do with CFEngine. There may not always be an "easy" way to do it, but if you can do it with commands on the OS, you can absolutely do it with CFEngine.

There Are, of Course, Drawbacks

No software is perfect, and CFEngine is certainly no exception to this. Because it does have more than two decades of history, there are some—shall we say—legacy pieces and behaviors that must be maintained intact for various reasons—usually the reason being someone who is paying a lot of money to CFEngine AS. It's a valid reason—that pays for it to stay open source, you know! But it can cause headaches on modern systems when you run into one.

It's also less a foot-shooting gun, and more a foot-shooting Gatling auto-cannon. There are certainly many ways to minimize the risk. However, eventually, you're going to make a mistake and not catch it. Most of the time, the only impact is the agents refusing to run your new policy because it's broken, and sticking with the previous policy. And sometimes it's rm -rf /$EmptyVariable/*—which will get run in parallel across your entire environment, usually in less than 5 minutes.

Because CFEngine is so powerful and flexible, it is also very easy to find yourself buried under the

possibilities. Seriously, they're just about limitless. But there are also just as many ways to end up with a sprawling mass of thousands of promises in hundreds of files. I've personally seen setups that rivaled /usr/src—but with a lot less organization. Keeping up with what you need to do and keeping the master files under control can feel like competing interests.

And one of the largest drawbacks by far is that FreeBSD is considered tier 2 by CFEngine. That doesn't mean they don't support it—far from it, thanks to the efforts of the port maintainers and a handful of us users. However, when it comes to function and feature, FreeBSD does not get all the goodies that Linux does. You aren't likely to run into anything you can't do, but cf-agent won't "automagically" report some monitoring data, and you'll have to spend more time writing your own promises and bundles since the standard promise libraries don't prioritize FreeBSD.

I'd Buy This for a Dollar! Where Can I Learn More?

Thanks to being one of the most mature and stable configuration management systems out there, CFEngine has an absolute mountain of resources available. The best place to start is the official CFEngine site (where you can also grab glossies to slip under the beancounters' doors) at www.cfengine.com. That's also where you can grab CFEngine Enterprise for the price of completely free (for up to 25 hosts). But as with any product this complex and powerful, that's just the beginning.

If you're ready to dive right in, Vertical Sysadmin offers a series of training videos and articles for the low, low price of completely free. With some help from whoever signs the checks, they also provide some of the best in-depth, one-on-one training you can get. In fact, I would recommend everyone interested in CFEngine start with their IT Automation with CFEngine: Business Values and Basic Concepts video.

However, let's be honest. Getting budget? Right. Not getting budget. For that, CFEngine has the help-cfengine mailing list. As you might expect, you'll see not only CFEngine Champions regularly providing assistance, but also CFEngine developers and employees. Just browsing through the archives, you'll often find the answer to your question is already out there. And there's also the #CFEngine IRC channel on Freenode.

Once you've got your first policy written and you're starting to get comfortable, I highly recommend reviewing the official Best Practices guides before you get too far along. While heavily geared toward Enterprise users, they cover everything from how you should use version control for your policy files to adjusting the policy for scaling to thousands of hosts.

Happy promising! ●

PHILLIP R. JAENKE is a systems engineer and administrator who also happens to do a bit more than dabble in storage, networking, and writing. He's been at it long enough to have written checks to Berkeley Software Design. When not busy keeping the lights on at large enterprises across a variety of Unixes, he chips into various open-source projects where he can, including CFEngine and FreeBSD. He also designs and engineers the widely-used BabyDragon VMware reference whitebox, and develops the TaleCaster comprehensive media system.