LUKE KANIES

Puppet

NEXT-GENERATION CONFIGURATION MANAGEMENT

Luke Kanies runs Reductive Labs, a startup producing OSS software for centralized, automated server administration (http://reductivelabs.com). He has been a UNIX sysadmin for nine years and has published multiple articles on UNIX tools and best practices.

luke@madstop.com

IN MY DECADE-LONG CAREER AS A system administrator, I have always done my best to use tools to lower my workload. I like to think one of my goals is to be the laziest person around, and great tools help me to reach that goal by allowing me to get more done with less effort.

I've spent the past few years as a consultant, integrating existing tools and developing new ones when the need arose. I generally restricted my development to smaller projects, because my revenue model did not allow me to take years or even months off for long-term development, which meant that I largely had to rely on existing tools as the primary solutions I provided. Experimentation with different ways of getting better tools, including contributing to existing projects and working within a larger organization, finally led me to attempt to create the tool that I really wanted to use to do my job. I call this tool "Puppet"; I have released it under the GPL, and I am building a company, Reductive Labs, around developing and supporting it.

Puppet is being developed with two purposes in mind. The first and most obvious purpose is to be the best configuration management tool available, such that I can build a company and community around making sysadmins' lives easier. Less obviously, I am developing Puppet to be an operating system abstraction layer (OSAL), something that functions as a cross-platform API into the features of the OS without forcing you to delve into the messy details that come with each separate distribution and release. I believe that providing this OSAL is a required step in providing the best automation tool, and I hope that other tools can also begin writing to this OSAL instead of coming up with a new way of handling each OS's messy details.

To download Puppet or the Puppet source code, go to http://reductivelabs.com. There you will also find links to the blog I maintain about Puppet's development, mailing lists, and everything else you've come to expect from open source projects.

A Low-Level Abstraction Layer

At some point all automation tools must isolate their users from detail, else those tools would take more effort than they saved. The point at which they hide detail, though, varies widely among tools and has a large impact on how those tools are used and developed. Tools like cfengine hide almost no detail at all, enabling the user to choose exactly how to interact

with the system but requiring that the user know too much about each supported platform, such as where the crontab command is or what command to use to add users. Conversely, tools like SmartFrog and LCFG use large, coarse-grained modules responsible for large swathes of functionality; these hide almost all detail, but they leave the user only as much room for customization as the module developer thought appropriate.

Puppet could be said to be either lower- or higher-level than cfengine, depending on how you stack it. On the one hand, Puppet is designed to hide details such as file locations and the differences between "useradd" and "adduser," which makes it higher-level than cfengine, but it also provides hooks to directly modify a much larger range of detail on a given operating system, so it could be said to be lower-level. I like to think of it as just being different: cfengine provides a simple API to the elements that the operating system cares about (files and file contents, packages, processes, etc.), while Puppet exposes an API to the elements that humans care about (users, groups, cron jobs, virtual hosts, etc.). There is some crossover, such as humans sometimes caring about file permissions, but more often than not two otherwise equivalently functional cfengine and Puppet configurations will look quite different.

I have ambitious goals for Puppet's OSAL. Package management systems could write directly to it, instead of each pre- and post-install script having its own idea of how to create a user or cron job. System administrators could use it to replace unstable or outdated tools without affecting the core functionality, as long as the OSAL could configure each tool equivalently. In the end, I hope that Puppet's OSAL will be the standard repository for all of the details necessary to configure each of the different operating systems.

A Better Automation Tool

Puppet is more than an abstraction layer, though. It is a whole declarative configuration management framework. In addition to altering the level and type of detail that administrators must handle, Puppet also raises the bar in terms of expressiveness and communication. It includes a simple but powerful language (LISP taught us that language power often comes through simplicity rather than complexity or variety) capable of expressing the relationships between the different elements of an operating system, along with a set of clients and servers meant to make it easy to get information into and out of your network.

A LANGUAGE BASED ON RELATIONSHIPS

Puppet's language is declarative, meaning that you specify the "what" (objects and their values) but not the "how" (how to turn those objects into configuration state). The OSAL library takes care of the "how" for us, so the language can focus entirely on the objects, their values, and how they all relate.

Puppet's language has only one goal: to get your network to provide the features you need in the way you need them. Given an abstraction layer that handles all the details of the different operating systems and applications, service provision turns into a process of collecting the list of objects you need, marking how those objects are related, and filling in the details. To provide an Apache service, you need to collect the package, the content to provide, the configuration for the service, the service itself, and maybe an IP address and a file system.

Puppet's language provides a classing mechanism to indicate that this collection of objects functions as an Apache service, and it goes one further by allowing you to specify the relationships between these objects, such as the obvious fact that the service can only actually be started when all of the other objects are cor-

rectly in place. It even allows you to go further still and specify whether the service should be restarted if the package gets upgraded or the configuration gets modified.

Puppet provides basic abstraction mechanisms in the language, so you can vary individual details based on other details (e.g., different groups for different operating systems) or vary the work itself based on details (e.g., provide different elements on different operating systems).

REUSABLE CODE

There is nothing resembling a CPAN for system administration tools, because we have never had tools that could separate these relationships that we care about—the elements that make up a service, how those elements relate to each other, and what details the elements should have—from the specifics of how to implement those relationships on a given platform and in our environments. The existence of an OSAL provides us the opportunity for that separation, so Puppet's language has been written with a focus on reusability. If you know exactly what it takes to make a Solaris 10 server secure or to provide Apache plus mod_perl, you can write a server class that does this and then share that class. When I have development time, I plan on creating community space online to facilitate this sharing, but I think Puppet's simple language will encourage it whether I facilitate or not.

COMMUNICATION IS THE KEY

Every sysadmin knows that tools cannot be silos, meaning they cannot be cut off from the rest of the tool fabric, yet for some reason most of our tools aren't very fond of talking to each other. Each tool seems to want its own user and host database, and it is often prohibitively difficult to get data from one application to another. This extra overhead usually just means that we hack up our own, simpler tools that all talk to each other but are too specific (and too embarrassing) to share, rather than using the well-known tools.

I will not let Puppet fall victim to that. I am building simple APIs into both the client and the server, using XML-RPC over SSL, so if you want to write a querying tool, you can, or if you want to replace some portion of Puppet with a much better tool that responds to the same interface, you can. In addition, I am doing everything I can to get data back out of Puppet—if you provide Puppet with information, it will do everything it can to take advantage of it, and you should certainly never have to tell Puppet the same thing twice.

I plan direct integration with the different tools in the sysadmin fabric, although I have been so focused on the core functionality (which is now release-worthy) that I have had little time to spend on this. Puppet will directly open tickets and configure your monitoring and trending systems, rather than assuming that you will just do that yourself, and when it does so it will provide everything it can to make your job easier. You have already given Puppet enough information for it to manage your network; it would be downright offensive if Puppet did not use that information to provide context to the information it gives you back. Every network has a somewhat constant stream of failures; the context of the failure, such as the services it affects and the overall failure rate, is what determines whether it is a critical failure or not, and Puppet will do everything it can to provide that context.

Puppet already has a built-in log centralization mechanism, for instance, and the logs that Puppet produces include the configuration path to the specific element that emitted the log, so you know whether a package installation failure is affect-

ing DNS or GDM, whether it's on a workstation or server, and whether it is on your main LAN or in your DMZ. You do not have to explain to the log server where your LAN is versus the DMZ, because you already explained that to Puppet, which just sends the information along.

By the way, Puppet does use SSL certificates for all authentication and encryption, but because of the inherent complexity in managing those certificates, Puppet includes a simple certificate manager to help you. By default, the central Puppet server creates a new certificate authority, each new client asks that authority to sign its certificate request, and there is a simple command-line tool you use to sign those requests.

The Syntax

Although there is a lot more to Puppet than its language, and it is expected that mechanisms other than the language will eventually be supported for input, the language is currently the only way to speak to the OSAL.

I have tried to keep Puppet's language as simple as possible. It looks more like a data dump than a language, and I plan on keeping it that way as long as I can. It only supports a few statement types, mostly centered around describing and aggregating objects, along with a few simple operator-like constructs for greater abstraction.

VARIABLES AND ASSIGNMENT

One of the statement types you will immediately recognize is assignment:

```
$variable = value
```

Simple words do not need to be quoted in Puppet (strictly speaking, words which match /[-\w]+/ in Ruby do not need to be quoted). Variable scopes work as one might expect (well, at least they work as I expect)—variables are visible when defined in the current scope or any enclosing scope. The only twist is that, in an attempt to be declarative, variables cannot have their values set more than once in a given scope.

When retrieving its configuration, a Puppet client collects a configurable set of facts about itself, and these facts are defined as variables in the top-level scope. The most useful facts are things like $operatingsystem (usually the output of uname -s), $ipaddress, $domain, and $hostname. These facts are all retrieved using a separately maintained library imaginatively named Facter (which you can get independently from http://reductivelabs.com), and Puppet converts them all to lowercase.

ELEMENT SYNTAX

Puppet's elements can be thought of as a kind of named hash, or named associative array. Any specification of low-level elements requires the element type and the element name, and all element types support a fixed list of arguments. When applied appropriately, the following snippet will verify (and fix, if necessary) the metadata of /etc/passwd and make sure that the latest version of the sudo package is installed:

```
file { "/etc/passwd":
    owner => root, group => root, mode => 644
}
package { sudo: install => latest }
```

You could use the stand-alone puppet executable to apply this snippet, and it would check that /etc/passwd is owned by user and group root, that its mode is

644, and that the sudo package is installed and is the latest version available (via whatever mechanism is defined to retrieve packages). Puppet will automatically fix any deviations it finds, although this can be set to just log deviations, rather than fixing them.

The "package" statement is a bit special, because Puppet knows what the default package type is for every platform on which it runs. For those platforms like Debian and RedHat that support automated retrieval of packages and their dependencies, this statement would be enough. For those platforms like Solaris and AIX that do not, you would need to provide additional information (e.g., a URL) on how to retrieve the package, but not how to install it or any of the messy details of putting it into a /tmp and so on.

You can easily specify multiple objects at a time, either separating them with semicolons or just using an array as the object name:

```
file {
    "/etc/fstab": owner => root, group => root, mode => 644;
    "/etc/named.conf": owner => root, group => named, mode => 644
}
```

This will check both of these files as though they had been specified in separate file blocks. This type of syntax is useful for those cases where you have many objects of the same type but with entirely different details.

```
file { ["/etc/shadow", "/etc/sudoers"]:
    owner => root, group => root, mode => 440
}
```

This will check that the two specified files have the exact same metadata. This is obviously useful for those relatively rare cases where you have many objects of the same type and with the same details.

You can, of course, combine them:

```
file {
    ["/etc/shadow", "/etc/sudoers"]:
        owner => root, group => root, mode => 440;
    "/etc/fstab": owner => root, group => root, mode => 644;
    "/etc/named.conf": owner => root, group => named, mode => 644
}
```

This is just a combination of the other two snippets, in the tersest (and, thus, not necessarily the most readable) form. Actually, you could get even more terse if you desired, but you'll have to see the documentation for how to do that.

White space does not matter, so any differences in it in this code are to help you, not Puppet.

CLASSING

Puppet supports three main encapsulation constructs: classes, components (created using the define keyword), and nodes. All three constructs are just named collections of objects, with some use-appropriate behaviors tacked on.

The simplest and most common construct is a basic class; it is useful for collecting a set of related elements which all get applied to provide a certain service:

```
class apache {
    package { apache: install => latest }
    service { apache: running => true, requires => package[apache] }
}

include apache
```

This class contains two statements, one that makes sure that an Apache package is installed and another that verifies that the Apache service is running (Puppet

defaults to using init scripts to start, stop, or check services) and also specifies that the service depends on the package. Once this class is created, it can be applied using the include keyword in any host's configuration. Classes also support inheritance, although this is more useful for defining server classes than for providing a specific service:

```
class base {
    package { sudo: installed => latest }
}

class webserver inherits base {
    package { apache: installed => latest }
}

class dnsserver inherits base {
    package { named: installed => latest }
}
```

This defines three classes, each of which verifies that a package is installed. Including any of these service classes also includes the base class, just as you'd expect inheritance to work.

NODE CONFIGURATION

Puppet provides a class-like structure for specifying a given node's configuration:

```
node kirby {
    include $operatingsystem, webserver
}
```

The include function can handle variables just fine, and kirby happens to be a Solaris x86 server in my basement, so when kirby connects, the central Puppet server will return a configuration containing all the work associated with the sunos and webserver classes.

CODE REUSE

The last organizational structure in Puppet is analogous to a function in other languages; I alternately call them components or definitions, and they are created using the define keyword. You would use it to specify a chunk of work that will be applied multiple times in the same configuration. For instance, I store many of my configuration files on a central server, so I have created a remotefile keyword that encapsulates all of the details that get repeated with each copy:

```
define remotefile(source, mode => 644) {
    file { $name:
        owner => root, group => root, mode => $mode,
        source => "/nfs/files/$source"
    }
}

remotefile { "/etc/sudoers": mode => 440, source => "sudoers" }
```

This defines and then uses the component remotefile, which is just a wrapper for a simple file statement. This file statement verifies that the local copy of the specified file is the same as the remote copy of the same file (I have chosen to use an nfs-mounted file, but Puppet supports some remote protocols, too) and then verifies that the local file has all of the correct metadata.

Component prototypes define the arguments that they accept, just like other functions, and Puppet components can have defaults. The only quirk in Puppet is that the value before the colon in a Puppet statement (e.g., /etc/sudoers in the

example) is an implicit argument and is set as the $name variable inside the component.

ABSTRACTIVE OPERATIONS

Puppet only has one operator in the strict sense of the word, and it only has one construct that resembles a control statement. The operator resembles a C-like trinary operator, modified (some would say butchered) slightly to fit in a bit better. It is useful for using one value to determine another value:

```
# remember $operatingsystem is set by Puppet for you
$fstab = $operatingsystem ? {
    sunos => "/etc/vfstab",
    default => "/etc/fstab"
}
```

Notice the use of the default value here as a fall-through option. These statements can also be used inline anywhere a value is needed, and multiple, comma-separated values can be provided for a given option. The other statement type useful for abstraction is a switch-style statement like many others; see the documentation for details.

Conclusion

Puppet is an ambitious new open source configuration management framework. It provides an extensible operating system abstraction layer (OSAL), along with a language that writes to that language and a set of clients and servers for sharing information such as configurations and file contents across the network. Puppet is still in its early stages and does not yet handle the majority of the elements that you need to manage, but it is easy to extend and has a keen focus on supporting and encouraging community involvement.