

# Making System Administration Easier by Letting the Machines Do the Hard Work, Or, Becoming an Agile Sysadmin

JOSHUA FISKE



Joshua Fiske is the Manager of User Services at Clarkson University. In this role, he tends the University's fleet of

Linux and VMware servers, handles day-to-day operations of the network, and manages a team of professionals who provide direct end-user support. He is passionate about the use of open-source software in higher education.

[jfiske@clarkson.edu](mailto:jfiske@clarkson.edu)

One of the challenges faced in the information technology field is the need to be responsive to the needs of our customers. To a system administrator, this typically manifests itself in one of two ways: responding to reports of broken stuff and responding to requests for new stuff. In this article we will focus mainly on doing the latter by taking advantage of virtualization, automation, and patch management.

In the days of yore, responding to a request for new stuff was a costly and time-consuming process that went something like this: Quote and order new hardware. Wait 30 days for hardware to arrive. Rack-mount hardware and install operating system. Configure operating system and services the same way “all the others” are. And don’t make a mistake lest we end up with a configuration that is different from everything else in our environment. Heaven help you if you need to make a change to a configuration or if you have a team of sysadmins who each have their “own way” to do a task. Two months later, you might have the new service up and running.

And where are users while you’re doing all this? They’re working on their own to find a way to meet their need *right now*. By the time you’ve worked your way through this process, they’ve found another tool or their needs have changed. All because you weren’t able to be agile in responding to their request. But what’s the alternative? Imagine being able to receive a request from a user and have a system standing and ready to service their needs in the same day (or 15 minutes later)! Here’s how I’ve been able to do just that.

## The Solution

To solve this problem, we take a three-pronged approach: virtualization, automation, and patch management.

By now, most sysadmins are familiar with the advantages of virtualization. Combining systems to run on common hardware ensures efficient use of resources, improves disaster recovery posture, etc. And in this case, it also allows us to be more agile in responding to requests for new systems or services. Because we have a virtualized infrastructure, we generally have some “extra” capacity to stand up new systems without needing to order hardware.

Once we have a virtualized system, we must install an operating system. This process of installing the operating system and configuring services is the largest opportunity for human error to affect the quality of our output, because each sys-

admin has a slightly different way of building servers and configuring services. We can fix this by leveraging tools like kickstart and Puppet to automate the process of building systems and configuring services.

And then once you have deployed a large fleet of systems, it can be very helpful to know which systems are in need of patching. A tool like Spacewalk can provide a centralized reporting interface to track this information and to push out updates.

## The Practical Solution

Knowing the theoretical solution is good, but I always find a practical example to be much more helpful. So, I'll walk you through the process of provisioning a new server.

We have a pretty standard virtualization environment. In our shop we use VMware ESX (and in some cases, ESXi) with a shared iSCSI storage environment. To provision a new server, we just log in to a VM host with excess capacity and create a new guest. We have a standardized disk size of 20GB for our Linux guests. If we need more storage, we can attach another virtual drive and, through the magic of LVM, we can make that space available to the OS. Once the system is defined, we attach an ISO image and set it as the primary boot device. But don't power on the system just yet.

This is where the exciting stuff begins. We start by creating a kickstart file to define the properties of the new system. Since we've done this before, we can just copy an existing kickstart file and make a few tweaks to it, setting things like the IP address and hostname. Then, we boot the system and respond with the following at the Linux boot prompt:

```
boot: linux ks=http://shep.clarkson.edu/provisioning/newhost.ks
ip=128.153.5.60 netmask=255.255.255.0 gateway=128.153.5.1 dns=128.153.5.254,
128.153.0.254
```

A full copy of our standard kickstart file is available linked from [www.usenix.org/login/2011-02](http://www.usenix.org/login/2011-02). The most interesting bits of that file are in our postscript, which is where we begin to automate the customizations that we generally perform at our site. Of course, you'll want to customize this to suit your environment. This is an opportunity to be a creative sysadmin; think about the tasks that you generally perform as part of your build process and think about scripting them here:

```
%post
#
# Disable some services
/sbin/chkconfig --level 123456 cups off
/sbin/chkconfig --level 123456 yum-updatesd off
/sbin/chkconfig --level 123456 apmd off
# Install Spacewalk
/bin/rpm -Uvh http://spacewalk.redhat.com/yum/1.0/RHEL/5/i386/spacewalk-
client-repo-1.0-2.el5.noarch.rpm
/usr/bin/yum -y install rhn-setup yum-rhn-plugin
/usr/sbin/rhnreg_ks --serverUrl=http://spacewalk.clarkson.edu/XMLRPC
--activationkey=<activation key here> --force
/bin/sed -i '/enabled=0/ d' /etc/yum.repos.d/CentOS-Base.repo
/bin/sed -i 's/\\(gpgcheck=.*)\\1\\nenabled=0/g' /etc/yum.repos.d/CentOS-
Base.repo
```

```

/bin/rpm --import http://mirror.clarkson.edu/rpmforge/RPM-GPG-KEY.dag.txt
/bin/rpm --import http://mirror.clarkson.edu/epel/RPM-GPG-KEY-EPEL
#
# Install Puppet
/bin/rpm -Uvh http://shep.clarkson.edu/provisioning/software/puppet/
factor-1.3.7-1.el5.rf.i386.rpm
/bin/rpm -Uvh http://shep.clarkson.edu/provisioning/software/puppet/
puppet-0.22.4-1.el5.rf.i386.rpm
/bin/sed -i 's/#PUPPET_SERVER=.*PUPPET_SERVER=shep.clarkson.edu/g' /etc/
sysconfig/puppet
/bin/sed -i 's/#PUPPET_LOG/PUPPET_LOG/g' /etc/sysconfig/puppet
/sbin/chkconfig --level 35 puppet on

```

Several things happen as part of our post-installation process. Reading through this code, we see that it configures some system services, joins the system to our Spacewalk server, and installs Puppet (and factor, which is a prerequisite for Puppet). Then the machine reboots and greets us with a login prompt.

As part of the Puppet setup, the client has generated a set of SSL keys and submitted them to the Puppetmaster server for signing. To allow the Puppet client to retrieve configurations and files from the Puppetmaster server, we need to sign that request. This is done on the Puppetmaster server with these commands:

```

/usr/sbin/puppetca --list
/usr/sbin/puppetca --sign <fqdn>

```

Now that our Puppet client is approved, it looks to the Puppetmaster to see what it should do. Our Puppetmaster directs the client to perform a number of actions, including creating users, sudoers configuration, installation of Apache, and some other settings.

## Creating Users

We create a predefined set of administrative users on each host, ensure that these users are members of the wheel group for sudo, and set an RSA key for SSH:

```

class users {
  user { "jfiske" :
    ensure => present,
    uid => 500,
    gid => "wheel",
    managehome => true,
  }
  ssh_authorized_key { "jfiske-key" :
    require => User['jfiske'],
    ensure => present,
    key => "<rsa_key_here>",
    type => "ssh-rsa",
    user => "jfiske",
  }
}

```

## ***Sudoers Configuration***

We can also push an entire configuration file to each host. In this case, we have Puppet push the `/etc/sudoers` file to each system to ensure that the user we created above also has sudo abilities. This is particularly useful if you have a team of administrators who each need to have the same level of administrative access to each system (without needing to distribute a system's root password):

```
class sudo {
    file { ["/etc/sudoers":
        owner => "root",
        group => "root",
        mode  => 440,
        source => "puppet://shep.clarkson.edu/configuration/sudoers",
    ] }
}
```

## ***Installation of Apache***

We can also install a software package and ensure that it is running. In this case, we install a Web server:

```
class apache {
    package { [httpd: ensure => installed ]

    service { [httpd]:
        ensure => running,
        require => Package["httpd"],
    ]
}
```

## ***Other Settings We Push***

We use Puppet to install packages, distribute configuration files, and ensure that services are running. We can even combine these three functions with dependencies to perform more complex tasks. In this example, we tell Puppet which packages should be installed for SNMP, push the relevant configuration file (`/etc/snmp/snmpd.conf`) and a binary file (`/usr/local/sbin/ntp_check`), and then ensure that the SNMP service is running.

```
class snmpd {
    case $operatingsystem {
        CentOS: { $snmpd_packages = ["net-snmp", "net-snmp-lib"] }
    }

    package { $snmpd_packages: ensure => installed }
    file { ["/etc/snmp/snmpd.conf":
        owner => "root",
        group => "root",
        mode  => 644,
        source => "puppet://shep.clarkson.edu/configuration/snmpd.conf",
        require => Package["net-snmp"],
    ] }

    file { ["/usr/local/sbin/ntp_check":
        owner => "root",
        group => "root",
    ] }
}
```

```

        mode => 755,
        source => "puppet://shep.clarkson.edu/configuration/ntp_check",
        require => Package["net-snmp"],
    }
    service { snmpd:
        name => snmpd,
        enable => true,
        ensure => running
    }
}

```

Once we have our Puppetmaster configured to perform all of our post-installation configuration steps, we can take the total system build time down to something around 10 minutes. If we are just adding a new application server to an existing pool, then we are done (because, of course, we've also Puppet'ed the install and configuration of the app server software). If we are deploying a new application, then we have a platform on which to begin developing/installing. Reducing system build time allows system administrators to begin to pull themselves up out of the daily minutiae of building systems and instead focus on the more important and interesting aspects of keeping their users happy.

### Footnote: Improved Management Abilities

Also, consider the benefits that are yet to be reaped with respect to patch and configuration management. Because we've joined the system to our Spacewalk instance, we will receive notifications when there are package updates that need to be installed. And because we've pushed configurations using Puppet, when we need to make global changes we can update one file and have it propagate to all of our servers. We should also consider storing our Puppetmaster's configuration files in a revision control system, so that we have a solid history of what changes were made when and by whom.

### Resources

Installing Spacewalk on CentOS: <http://wiki.centos.org/HowTos/PackageManagement/Spacewalk>.

James Turnbull, *Pulling Strings with Puppet*: <http://apress.com/book/view/1590599780>.