O'REILLY® Short Cuts

Managing RPM-Based Systems with Kickstart and Yum

by Ethan McCallum

ISBN: 978-0-596-51382-5

Released: March 12, 2007

Managing multiple Red Hat-based systems can be easy—with the right tools. The yum package manager and the Kickstart installation utility are full of power and potential for automatic installation, customization, and updates. Here's what you need to know to take control of your systems.

Contents

Manual Versus Automated

Why Kickstart and Yum?

Kickstart and `yum` are tools to automate installation, updates, and upgrades of Linux systems. Specifically, they operate on RPM-based Linux variants, such as the Red Hat Linux line (including the Enterprise and Fedora branches) and CentOS. Both tools have a learning curve and require some infrastructure for you to use them to their fullest extent.

As with any tool, it's fair to ask, "What can it do for me? Why would I take the time to learn it?" My inspiration was, well, installing Fedora Core several times. To see what I mean, consider some of the information you enter when you install a Red Hat operating system (OS) by hand:

- Choose your install type (initial install or upgrade).
- Carve out your disk structure.
- Configure networking.
- Define a basic firewall.
- Choose time zone.
- Set the root password.
- Configure the boot loader.
- Choose which software to install (use the predefined groups or cherry-pick individual products).

This list covers only the base install, too. After that you typically install third-party RPMs, apply some local customizations, and call `yum` to apply any OS updates ("patches" for the old-school among us).

Now, consider the equivalent Kickstart install. After you boot from the CD, enter:

```
linux ks=http://your.install.server/ks.cfg
```

and walk away.

To me, the latter method looks cleaner, more elegant, and less involved than its counterpart. Instead of interacting with the OS installer, I feed it a configuration file—that's what the URL with the *ks.cfg* is all about—with answers to all of its questions. Disk layout? In the file. Software to install? In the file. Time zone? In the file. You get the picture. In certain cases, I can use the same config file for many machines.

Automation tends to win out over manual processes for three core reasons, and Kickstart is no exception:

Consistency

The manual install method involves lots of human-computer interaction, which is tedious at best and error-prone at worst. If you split the workload between two people, rest assured you'll be able to tell who installed the OS on a given machine.

Time

If some other task holds your attention—maybe you're fighting production fires while setting up some new servers—the manual install can stretch over several hours while the machine waits for you to click an OK or Yes button. Initiating a Kickstart install, by comparison, takes a few seconds. As long as you get the URL right, you type it once and walk away.

Cost

If you're a senior sysadmin, your company makes better use of your time (and their money) when you handle high-level architectural work. Machine setup is best left to junior staff members, but if they make a mistake you know you'll have to help clean it up.

Overall, Kickstart does for OS installs what robots and conveyor belts do for factories: it removes the human element from the process. While the machine takes the dull, repetitive work—which computers do willingly and very well!—the human minds can focus on those tasks that require thought and innovation.

Is This Too Good to Be True?

Consider another fair question: "Is this tool right for me?" Depending on your situation, maybe or maybe not.

Establishing a Kickstart environment requires some infrastructure and effort. Expect even more work if you want to add change control to your automated `yum` updates. While most of this is up-front, one-time work, you need enough OS installs ahead of you to make Kickstart worth your time.

Some math will illustrate my point. Suppose that a fairly vanilla manual install takes one hour. Creating and testing a simple Kickstart environment can take a full eight-hour workday, plus an additional 5 or 10 minutes per machine if your hardware is not consistent. This means you reap any benefit on your Kickstart investment only after you have at least 9 or 10 (re)builds to do.

My oversimplified math doesn't account for a faster sysadmin who establishes a Kickstart environment in half a day, nor any cleanup required to correct mistakes made in a manual install. All the same, the hobbyist who installs the OS two or three times a year won't see the same benefits as the person who has to build a rendering farm before Monday.

I emphasize the terms "builds" and "installs" and not "machines." In a corporate test lab or student computing lab, in which the machines are regularly refreshed to a pristine state, a nightly refresh schedule adds up to 30 rebuilds per month, *per machine*. With only five machines—150 installs per month—Kickstart certainly merits your attention.

Will I Be a Kickstart Guinea Pig?

One last question for evaluating new tools: "will I be alone in the world?" In other words, who else uses this tool and are their needs similar to mine?

It's tough to tell which mid- to large-sized shops use Kickstart. I can't imagine who wouldn't, but then again, it's a rare company that publicizes its internal infrastructure process.

That said, a quick web search uncovers several blog and wiki sites with Kickstart and `yum` sections, not to mention a couple of mailing lists. People using these tools are certainly not alone.

As an author of Kickstart and `yum` articles, I sometimes get success stories from readers. Academic institutions, small businesses, and corporate titans alike use these tools to automate their systems management. One reader in particular boasted about having Kickstarted an entire server farm before lunch. That's a prime example of how a large shop can leverage these tools to save countless work-hours and maintain consistency across builds.

I'm Sold. Where Do I Get Them?

That's the beauty: both Kickstart and `yum` are included in the base Red Hat (and Fedora, and CentOS) operating systems. The tools aren't part of the default install —I'll explain later what to do—but rest assured that, as long as you have an install CD, they're at your fingertips.

How Do I Work All of This Magic?

Keep reading! In the coming pages, I'll show you how to prepare an infrastructure around Kickstart to rebuild similar machines with very little human intervention. I'll also show how, with a little more work and some extra tools, you can wrap `yum` cronjobs in a blanket of change control to prevent surprise updates. Mix a few

gigabytes of storage and some scheduled jobs, and your machines will be one step closer to managing themselves.

Automating the Build

This section explains some Kickstart basics and how to run your first automated build. If you have a massive server build-out due by Monday morning, this chapter is for you.

Ingredients

Mixing a Kickstart cocktail requires a *target machine*, a *config file*, and *install media*. The target machine is the one you're building. The config file is a virtual you: it holds all of the values you would have manually entered, and answers to all of those questions posed by the installer. Install media is a fancy term for "all of the files on the installation CD or DVD."

You only have three dots to connect here, but Kickstart lets you decide how to draw the lines. You could say Kickstart is painfully flexible: it can load the config file from a CD or floppy disk, a USB key, or over the network via URL. In that last case, it doesn't even have to be a real file—it can be something created on the fly by a servlet or PHP script. The target machine can load the install media from a local hard disk, an HTTP or FTP server, or even an NFS mount. The only constant is the target machine, and some may argue even that needn't be real—virtual servers, such as VMWare guest systems, build without a hitch.

All of that flexibility means you can kickstart a machine in just about any situation. That also makes it difficult to describe a "typical" Kickstart install end-to-end. I will focus on HTTP (web-based) install media and config files, but drop notes on how to use other methods.

While it's not mandatory for a Kickstart install, you need a separate workstation at your disposal in almost every case I can imagine. You can use this to manage config files, read online docs, and play games while you run your first Kickstart tests. If you opt for network-based installs, this workstation can serve up the install media and even operate a DHCP service for the target machine.

Install Media

The easy part of the process is handling the install media.

Directory structure

All Red Hat OS installations, whether automated by Kickstart or manual via the Anaconda GUI, expect to find files in a certain structure. Consider the root of the first install CD:

```
{product path}
   |
   +--base
   |
   +--RPMS
```

The *product path* depends on the OS you're installing: *RedHat* for Red Hat Enterprise Server, *Fedora* for Fedora Core, and *CentOS* for CentOS. (Early versions of Fedora Core used *RedHat* as the product path.) Your boot media has to match the product you install. For example, if you boot a CentOS OS install CD, the product path cannot be *RedHat* or *Fedora*. The installer will (rightfully) claim it can't find its files and ask you to enter another location.

base/ holds some metadata files. I'll cover those in more detail later.

The *RPMS/* subdirectory contains, well, RPMs. An RPM is a special archive file that contains software packaged for Red Hat systems. Whether it's a mail client, web browser, or window manager, every software product included in Red Hat, Fedora, and CentOS install is bundled as an RPM. (The term "RPM" also refers to the package management system itself—"**R**ed Hat **P**ackage **M**anager"—as well as its command-line tool, `rpm`.)

Filling the required directory structure is straightforward. First, pick an area with enough space. This is typically two or three gigabytes per OS revision. You'll need more space to handle OS updates, but that's a few sections away. In case it's not clear, this is space on a machine other than the target machine.

If you have the install CDs or DVDs (or their respective ISOs), mount them one by one and recursively copy their contents to your local disk. For example:

```
## mount the ISO
# mount -oloop,ro /path/to/first/cd.iso /mnt

## copy the files. Ignore errors about "TRANS.TBL"
# cp -rp /mnt/* /some/other/local/path

## umount the ISO
# umount /mnt
```

There are many, many other ways to do this. If you're more familiar with `rsync` or even the classic `find`-piped-to-`cpio` routine, have at it. As long as you maintain the directory structure, it doesn't matter which command you use.

If you don't have the ISOs or CDs handy, that's not a problem. Pick your preferred download mirror from the Fedora or CentOS web sites, then run an overnight, recursive `wget` job to fetch the files. Consider this command line:

```
wget --progress=dot:mega --recursive
  --no-parent --relative
  --exclude-directories='*/SRPMS/*,*/debug/*'
  --no-clobber
  --directory-prefix=/where/to/put/files
   http://some.mirror.site/path/to/files
```

`wget` prints a period (or "dot") to show download progress. By default, one dot represents one kilobyte of data. For large files this can produce a lot of needless output, so the option `--progress=dot:mega` tells `wget` to print a dot for every 64 kilobytes of data. Per `wget`'s formatting, that means each line of output is three megabytes.

The next three switches work in tandem. `--recursive` tells `wget` to follow links in every page it downloads, recursively. `--no-parent` and `--relative` limit that recursive action, so your download doesn't branch off to other parts of that web site or even to other web sites altogether.

Some mirror sites like to put source RPMs or special RPMs with debug info under the main tree. The switch `--exclude-directories='*/SRPMS/*,*/debug/*'` tells `wget` to ignore those directories.

If something interrupts your `wget` job—ISP failures, a machine crash, or whatever—you can always pick up where you left off. When `wget` sees the `--no-clobber` switch, it skips files it has already downloaded. It can't catch incomplete files this way, though, so remove that last, partially downloaded file before you start `wget` again.

Finally, the `--directory-prefix` switch indicates where you want `wget` to put the downloaded files.

Serving it up

Having the install media properly laid out is only worthwhile if target machines can get to it. You can copy the files to an external disk that you connect to the target machine, but that limits you to one machine build at a time. I'd recommend serving the install media over the network using an NFS, FTP, or HTTP service.

HTTP's stateless nature makes it the most flexible and, arguably, the most scalable of the network methods. A simple round-robin DNS lets you transparently balance the load among multiple machines. With a little more DNS magic you can move the install media from one machine to another, and target machines are none the wiser.

In fact, the target machines don't really care what's on the other end of the connection, as long as it talks HTTP and doesn't return 404 errors when they request

files. That machine can serve your install media using Apache's httpd, LightTPD, or even your pet squirrel frantically hammering out signals in binary. The only catch is that the web server software must listen on the standard port 80, due to a limitation in Kickstart.

The web server machine's OS doesn't matter, either. A basic Kickstart HTTP service can run Linux, Solaris, or even something of the Windows family. Some of the more advanced Kickstart and yum techniques that I describe later will require a Red Hat-based OS, however.

As an added bonus, the install media tree isn't just for Kickstart installs. A manual install can also load the install media from a network source. Having an internal install media service means you can build machines from any location in the office, without having to tote around a series of CDs.

Creating the Config File

The Kickstart config file can go by any name, but for simplicity I call it *ks.cfg*. This file tells the OS installer how to configure the target machine (such as setting the time zone and root password) and what software you want to install (that is, the oodles and oodles of RPMs).

You have a choice on how to create *ks.cfg*. (There goes that Kickstart flexibility again....) A manual install creates a *ks.cfg* for you. You can copy and modify the file */root/anaconda-ks.cfg* from a freshly built machine. This technique works best if you're building lots of similar machines (think "clone farm") or rebuilding the same machine several times over ("computer lab"). Reviewing an existing file can also help you learn *ks.cfg* syntax.

Barring that route, Red Hat, Fedora, and CentOS all ship with a tool called *Kickstart Configurator* (Figure 1). Click through this GUI and it will create a *ks.cfg* befitting your input. Kickstart Configurator isn't part of the default install, though, so you may have to install the `system-config-kickstart` RPM before you can use it.

Then run:

```
# system-config-kickstart
```

to launch the tool.

If you're running an older OS, such as Fedora Core 1, both the product and the command are `redhat-config-kickstart`.

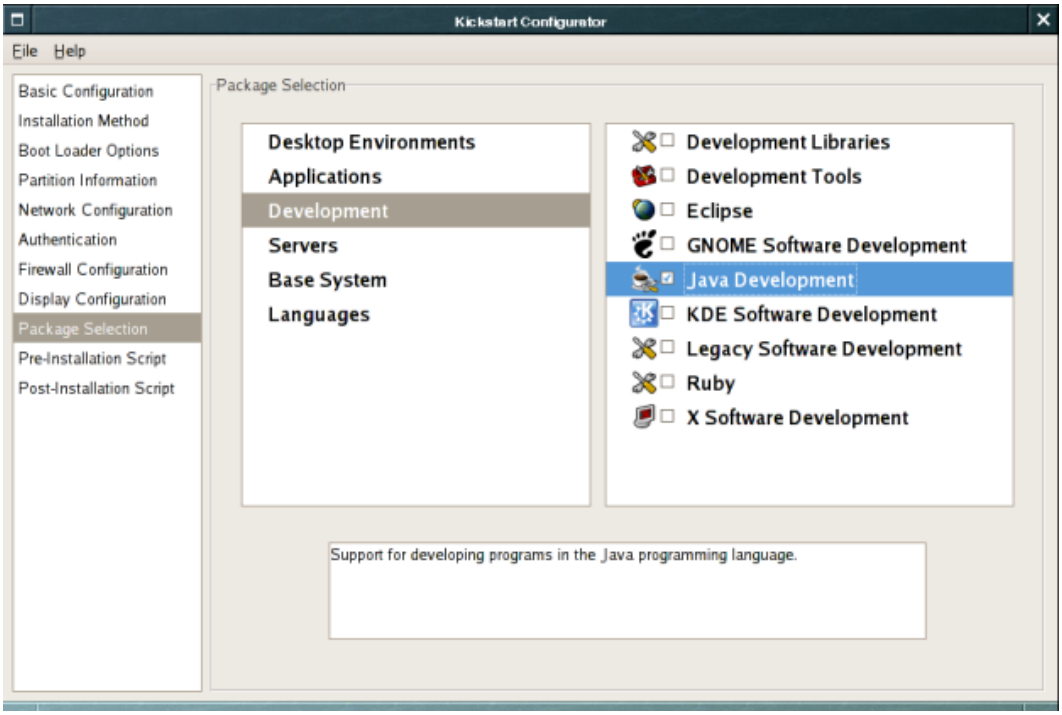

Figure 1. A glimpse of Kickstart Configurator

If you have memorized the Kickstart documentation—and no, I haven't—then another option is to create *ks.cfg* from scratch. Fire up your preferred text editor and get to work.

Whatever your route, it helps to be at least mildly familiar with the *ks.cfg* file format. I provide a brief rundown of the directives in the last section. In the meantime, Example 1 is an annotated sample file.

Example 1. Sample ks.cfg

```
## this is an OS install (as opposed to an upgrade)
install


## fetch the install media from the HTTP service
## running on os-webdist.internal
url --url http://os-webdist.internal/FC5-i386-install


## language support, for the install and the final OS, respectively
lang en_US.UTF-8
langsupport --default en_US.UTF-8 en_US.UTF-8


## basic keyboard and mouse settings
keyboard us
```

```
mouse generic3ps/2 --device psaux


## configure the video card
xconfig --card "VMWare" --videoram 16384 --hsync 31.5-37.9 --vsync 50-70
 --resolution 800x600 --depth 24


## network: the device eth0 will have a static IP address of 10.10.10.237
network --device eth0 --bootproto static --ip 10.10.10.237
  --netmask 255.255.255.0 --gateway 10.10.10.246
  --nameserver 192.168.100.201
  --hostname fedora-test


## you can also provide an encrypted password
rootpw ThisIsPlainText


## No firewall yet.  We can manually set that up later.
firewall --disabled


## use shadowed, md5-hashed passwords
authconfig --enableshadow --enablemd5


## local time zone
timezone Europe/Paris


## install the boot loader in the MBR
bootloader --location=mbr


## destroy all disk partitions, and initialize any
## disks that have no partitions
clearpart --all --initlabel


## setup /boot as its own disk slice, then
## put everything else under LVM's control
part /boot --fstype ext3 --size=75 --asprimary
part pv.00 --grow --asprimary

volgroup vg00 pv.00
logvol / --vgname=vg00 --size=2048m
logvol swap --vgname=vg00 --size=256m
```

```
## Which RPMs should Kickstart install?
## Lines that start with "@" are predefined groups, which
## I'll explain later.  Everything else is an individual
## RPM

%packages
@ dialup
grub
kernel
```

Connecting the Dots

After all that hard work, it's time to enjoy the fruits of your labor: get the client to read *ks.cfg* and load the install media.

Boot!

First and foremost, your target machine must boot the OS installer. The most common way to boot is by CD or DVD. You don't have to download the entire ISO for the first install CD, though. The file *boot.iso* (under the *images/* directory) weighs in at just a few megabytes and packs a simple boot image. Download it from your favorite Red Hat, Fedora, or CentOS mirror site and burn it to CD.

If you're running Linux or another Unix-like OS, chances are you can use `cdrecord` to write the *boot.iso* image to your CD or DVD media:

```
# cdrecord dev={device} data=boot.iso -v
```

where *{device}* is your CD or DVD writer device. Depending on your OS and hardware, this will be */dev/cdwriter*, */dev/dvdwriter*, or a device listed when you run `cdrecord -scanbus`.

If your CD-ROM drive has seen better days, boot from a USB key, a SCSI device such as a ZIP disk, or any other bootable media supported by your BIOS. Download the image `diskboot.img` from a mirror site (or copy it from the first install CD, if you have that) and write it to the device:

```
# dd if=diskboot.img of={device}
```

Replace *{device}* with the name of your removable device, such as */dev/sda* or */dev/sdc*. Be sure not to write to a *partiton* on that device, such as */dev/sda1* or */dev/sdc3*. That won't boot.

Do yourself a favor and double-check that device name. Then check it once more. `dd` sometimes stands for Disk Destroyer because it can render a disk's data unrecoverable. It is a woefully unforgiving tool, and will gladly do what you say even if that's not what you mean. Then, you'll say all sorts of things I can't put in print.

Some fairly modern target machines can forego physical boot media altogether and use technology called Pre-Execution Environment, or PXE (pronounced "pixie"). PXE is a fancy way of saying that the BIOS can boot off the network adapter and take instruction from a bootp or DHCP server, instead of loading an OS from disk. Setting up a PXE service is beyond the scope of this Short Cut, though a web search yields several online resources.

Load the configuration

Once the machine boots, it will greet you with the ever-familiar `boot:` prompt. Remember that one-liner command I showed in the earlier section Manual Versus Automated?

```
linux ks=http://your.install.server/ks.cfg
```

You don't necessarily have to serve *ks.cfg* from your httpd service. The target machine can load *ks.cfg* from any number of places. For one, you can put it on a diskette:

```
linux ks=floppy:/server4.cfg
```

(You can also substitute `fd0` for `floppy`.) This command tells Kickstart to use the file *server4.cfg* in the root of the diskette's filesystem. I found diskettes convenient for my early experiments (circa Fedora Core 1) but grew weary as I tried to build multiple machines. That, and diskettes had an awful habit of going bad on me.

Note that you can call the config file by any name. Here, I call it *server4.cfg*, perhaps to build a machine called "server4." You can store several config files on a single piece of media—they're at most a few kilobytes in size—and use whichever one befits the current target machine.

An alternative to the floppy is a SCSI or USB device, such as a thumb drive. Copy *ks.cfg* to that filesystem and enter:

```
linux ks=hd:{partition}:/ks.cfg
```

at the `boot:` prompt.

Here, *{partition}* refers to the device partition name, such as *sda1* or *hdc2*. The one strike against this method is that the device naming is not always predictable. If you have put *ks.cfg* on the only SCSI device in the system, it should appear as /dev/sda. (Remember, Linux sees USB disks as SCSI disks, hence the *sd* designation.) If there are other SCSI or USB disks in the system, however, it's a crap shoot. Be prepared to cycle through *sdb, sdc*, and so on until you find the right one.

You can also burn *ks.cfg* to a CD or DVD:

```
linux ks=cdrom:/ks.cfg
```

I mention this method just for completeness, though I advise against it; every time you change *ks.cfg* you'll have to burn another CD. You'll have several shiny coasters by the time you've ironed out the kinks in your config file.

The most flexible method is to serve *ks.cfg* over the network via HTTP, NFS, or FTP:

```
linux ks=http://your.install.server/ks.cfg
```

Replace `http` with `ftp` or `nfs` as appropriate. One benefit of the HTTP method is that *ks.cfg* needn't be a real file: you can point to a dynamic resource such as a servlet, PHP script, CGI, or whatever. As long as the httpd service returns syntactically correct, plain-text content, Kickstart doesn't care what creates the data.

Let it run

Whatever method you use to load *ks.cfg*, once it loads, it has loaded. Even a minimal install will take a few minutes to complete, and a watched pot never boils. Why not take a break while it runs? Expect this to be the first of many tea (or pub) breaks in your Kickstarted life. At least, until your manager realizes you're not really taking all day to build those new servers....

Troubleshooting

Full disclosure: my first Kickstart attempt didn't go over well. Nor did the next few. It didn't help that the error messages were cryptic or outright confusing. Some failures yield a stack trace from the underlying Anaconda installer's Python code. Tying that back to my *ks.cfg* or install media setup was tricky. I had to develop a toolkit for tracking down failures.

For one, every time the target machine asks for a file via HTTP, the web server records that request in its logs. Those logs are a goldmine of information. Successful file fetches (HTTP code 200) trace the install process, because they show you what files a target machine requests and when. If the web server can't find a file (HTTP code 404), chances are your tree of install media is out of place. By comparison, when the web server refuses to return a file (HTTP code 403), that indicates a file permissions problem in the install media tree.

Second, you can add the `interactive` directive to *ks.cfg* to step through an install. Think of this as a manual install, with your *ks.cfg* values as the defaults. (You must enter the root password manually, though.) If your config file is syntactically correct but there's a typo in a value—such as the wrong DNS server—the interactive mode is how to find it.

Pair up your interactive mode with a text-mode install. The GUI is a little easier on the eyes, but a text install lets you flip through the various virtual consoles to

peek behind the scenes. `Alt-F2` is a shell session. (Don't `exit` this; you can't get it back.) An Anaconda trace runs on the console at `Alt-F3`, and `Alt-F4` is a system-level trace of `mount` calls and `dmesg`-like output. Hit `Alt-F1` to return to the main screen.

When all else fails, try a manual install that points to your Kickstart install media. The GUI form of the installer may yield more useful error messages, or at least put them in a scrollable window so you can see the entire stack trace.

Customizing Your Kickstart Install

Kickstart takes a lot of the manual labor out of OS installs, which makes it a real time-saver. This convenience can make you lazy: after that first taste of a hands-off build, you want to further remove yourself from the process. Tweaking a *ks.cfg* or typing a URL at the `boot:` prompt can seem like a real chore.

This section has a series of customization tips to help you enhance your Kickstart experience.

Pre- and Postinstall Scripts

Kickstart will gladly install the base OS for you, but it's rare that your job ends there. More than likely, you have to further customize the machine to meet your (or your shop's) standards: create some canonical directory structures, install non-RPM'd software, or update config files.

Luckily for you, Kickstart also lets you run some code before and after it does its job in *pre-* and *postinstall* scripts, respectively. You get one of each, stored inline in *ks.cfg*, in sections labeled `%pre` and `%post`. (These sections get extracted into separate files and executed at build time.) For example:

```
## ... other Kickstart directives, such as the network config
## and root password ...


## postinstall script
%post

PATH=/sbin:/usr/sbin:/bin:/usr/bin

## disable printing daemons
chkconfig cups off

## create our special mount point area
mkdir /mnt/SpecialMount1

## disable the bundled yum repos
for file in /etc/yum.repos.d/*.repo ; do
```

```
        mv -i ${file} ${file}-DISABLED
done
```

Both the `%pre` and `%post` sections run under a plain Bourne shell (`/bin/sh`) by default. Append the `--interpreter` flag to specify a different interpreter, such as `/usr/bin/perl` or `/usr/bin/python`. Your main limitation is the tools available at the time the scripts are run, though: a preinstall script executes in the realm of the boot media, which includes a base toolset for rescue operations.

In all honesty, I've never needed to use a preinstall script, and I've rarely encountered them. The example in the Red Hat documentation—defining the partition scheme based on the number of disks reported in `/proc`—seemed slick at first, but I questioned how one would make the logic 100 percent foolproof. Some people use preinstall scripts to preserve data during a Kickstart upgrade (see the next section). In both cases I'd rather not let Kickstart handle this work for me—I'm happy to do it myself.

By comparison, the postinstall script runs `chroot`ed inside the freshly installed OS. It therefore has access to any tools you installed on the target machine. If your OS build included Perl or Python, then, your postinstall script could run code in those languages. As an alternative, you can specify `%post`'s `--nochroot` flag to not run the postinstall script inside the chroot. In this case, your postinstall script will have the same limited toolset as the preinstall script, but you then have the option of performing some other activities and manually invoking `chroot` to work inside the chroot area.

Postinstall scripts are plenty useful. Because they can operate on the newly installed OS, they can perform tasks you want to handle after the OS install but before the machine reboots into full-user mode. For example, you can call `chkconfig` to disable unneeded services:

```
for SERVICE in avahi-daemon avahi-dnsconfig cups ...etc... ; do
        chkconfig ${SERVICE} off
done
```

Pre- and postinstall scripts seem like quite a blessing at close range. From a higher-level, architectural view, however, they can grow ugly. It's easy to get tunnel vision and use pre- and postinstall scripts for tasks that other tools do better, all for the sake of doing it in the build. If your pre- and postinstall scripts do more than disable services or create directory paths, ask yourself a few questions:

Is this too complex?

Borrow a rule of thumb from larger software projects: keep it short and simple. Code that looks too slick now will probably be brittle tomorrow. Because pre-

and postinstall scripts run in the context of a Kickstart build, the only way to test a fix is to rebuild that test machine. In this case, would you rather track down a problem in a 10-line script or something that reads like a classic novel?

Can I do this better by hand?

It's a noble goal to offload work to Kickstart, but you'll eventually reach a point of diminishing returns. Try to weigh the time required to do something by hand, as part of each build, against the design, testing, and maintenance of a deep script. When your postinstall routine is calling an Expect wrapper so you can install that non-RPM'd vendor software....

Can I abstract this into a separate tool?

There's no rule that says your pre- or postinstall script can't call something else to do the heavy lifting. Trim your debug time by moving your logic into a separate script that you can run and debug outside of Kickstart. In turn, your `%pre` and `%post` sections then become one-liners to invoke these external scripts.

Should I be doing this here?

Even if it works in Kickstart, is your build process the right place to create application-level user accounts or set an automounter configuration? Is there any chance you'd need to perform this same action on an existing host? Some tasks work much better with a formal change-management process, such as a homegrown rsync job or cfengine. These options are also automated, but they are much easier to debug than something that runs only at build time.

Is this future-proof?

Remember that the scripts run in a limited environment. You don't want to rely too much on outside services—be they a network drive or database—because then you have to hard-code paths, hostnames, and perhaps even passwords into your build process. Those can all change over time, and who wants an expired account to derail a set of Kickstart builds?

Custom package groups

A *package group* is a collection of related software products addressable by a single, convenient name. You may recognize some of the default package groups, such as "Development Libraries" and "Office/Productivity," from the install. Package groups are a shorthand in *ks.cfg* because you can specify them instead of the individual products therein:

```
@ PackageGroupName
```

Kickstart will install all of the products in that group.

You're not limited to the default groups, though. You can define your own. All it takes is a text editor and a passing understanding of XML. Under your base install tree, note the file *{product path}/base/comps.xml*, commonly known as the *comps file*. (The product path is typically one of `RedHat`, `Fedora`, or `CentOS`. The earlier section Manual Versus Automated describes the product path and the install tree structure.) Example 2 has an excerpt from a typical comps file.

Example 2. Excerpt from a typical comps.xml file

```
<comps>

  <group>
    <id>admin-tools</id>
    <name>Administration Tools</name>
    <name xml:lang="fr">Outils d'administration</name>
    .... other <name /> elements ...
    <description> ...description, in default language... </description>
    <description xml:lang="fr"> ... description, in French ... </description>
    <uservisible>true</uservisible>
    <packagelist>
      <packagereq type="default">authconfig-gtk</packagereq>
      <packagereq type="default">system-config-date</packagereq>
      <packagereq type="default">system-config-date</packagereq>
      <packagereq type="optional">system-config-kickstart</packagereq>
      ... other <packagereq /> elements ...
    </packagelist>
  </group>

  ... other <group /> elements ...
</comps>
```

A `<group>` element defines a package group. The `<id>` element specifies the name that will appear in `ks.cfg`, whereas `<name>` and `<description>` appear during a manual install's package selection windows.

`<name>` and `<description>` default to English. If the OS install is running in a different language—for example, if you chose "French" from the menu—the installer uses the `xml:lang=` attribute to find `<name>` and `<description>` elements befitting that language choice. The excerpt in Listing 1 can also show the name and description in French (`fr`).

Set the `<uservisible>` element to `false` to prevent the choice from appearing during an interactive install. It has no impact on an automated install.

`<packagelist>` lists the packages that are part of this group. Each `<packagereq>` element specifies a single product name. The `type` attribute tells you whether a given package is included by default (`"default"`), is not included but can be added

("optional"), or cannot be removed ("mandatory"). In a manual install, you can uncheck the packages labeled type="default" but not those labeled type="mandatory".

You now have the information to create your own custom group. However, before you continue, I recommend you make a backup of this file. For one, it never hurts to have the original around in case something goes awry. With that done, it's time to fire up that text editor.

First, create a new <group> element and give it a fitting <id>, <name>, and <description>. If you are feeling worldly, you can add names and descriptions in multiple languages.

Next, specify the <uservisible> element. It wouldn't hurt to make this true, so you can select the new group during a manual install. (Remember, a manual install creates a *ks.cfg* that you can use to build a clone farm.)

All that's left is to pick the packages. Do yourself a favor: don't rely on an RPM's name for its <packagereq> element. Instead, use the rpm tool to query the package and get the name specified in the RPM metadata:

```
# rpm -q--queryformat '%{NAME}\n' -p some_file.rpm
```

You don't need to worry about managing package dependencies, either. Pass the --resolvedeps flag to %packages, and Kickstart will add any prerequisite packages behind the scenes. If your custom group includes the postgresql-pl package, for example, Kickstart will resolve the dependencies of postgresql and postgresql-server and install those as well.

Add the group to edit *ks.cfg* and run a test build. If your group's <id> is WebServerHost, then put:

```
@ WebServerHost
```

in the %packages section of *ks.cfg*.

Why would you want custom package groups? From an aesthetics perspective, it's cleaner to specify @ OurStandardInstall in *ks.cfg* than to list 40 or 50 individual package names. From a systems management perspective, it's cleaner: you can create package groups to match your systems' roles. Suppose that you define a group called BaseBuild, which has the packages you'd install on every system. You also have groups called DatabaseHost, DeveloperWorkstation, and SysadminLaptop, which are related to machine roles. Now, you can eyeball *ks.cfg* to determine what sort of target machine it would build. For example:

```
%packages
@ BaseBuild
@ SysadminLaptop
```

would build a machine with software suitable for a mobile sysadmin—perhaps some network analysis tools and WiFi drivers.

You can also layer groups as needed. If you know in advance one developer will be writing native code tools as well as some Java-based GUIs, define fitting groups and include:

```
%packages
@ BaseBuild
@ JavaDeveloperWorkstation
@ CCplusPlusDeveloperWorkstation
```

in *ks.cfg* for his or her workstation.

In a higher-level view, custom package groups let you separate the role's *type* ("web server host") from its *definition* ("requires the `httpd` package, plus modules X Y and Z"). Over time you can tweak the RPMs associated with a particular group, but the name specified in *ks.cfg* remains the same. Your *ks.cfg* files automatically and transparently pick up the changes you make to the group definitions on the next build.

As a final tip, I strongly recommend you create *custom* package groups instead of changing the base groups. Even though all of the group definitions live in the same file, consider your definitions an extension of what's already there. That makes it cleaner to move the changes to a new comps file: you just have to grab your custom groups (hint: demarcate them with big, clear XML comments) instead of picking through the base groups for your customizations.

Custom RPMs

It's not uncommon for a site to install RPM-packaged software that's not part of the base OS. Expect a shop with Linux-savvy sysadmins to manage homegrown software, or local customizations of third-party tools, using RPMs as well. You can include those RPMs in the build process and have Kickstart install them for you, right alongside the software that's bundled with the OS.

This just requires another quick trip to the comps file. Create a new group, or add the package(s) to one of your custom groups, and you're ready to roll. Once again, use `rpm -q` to get the package's internal name for use in the `<packagereq>` element. When you're done, run the `genhdlist` or `createrepo` commands to regenerate some metadata. I describe `genhdlist` and `createrepo` in more detail in the next two sections, Kickstart Upgrades and Custom Yum Repo.

Dynamic *ks.cfg*

I've made a few references to using a prototype *ks.cfg* to build clone farms or otherwise similar hardware. Kickstart does this very well, with one caveat: if the machines are the slightest bit different—every machine has a different hostname, right?—you have to tweak that prototype file for every build. Either that, or you copy the prototype, such that you have one file per machine. Heaven help you if you should get through the 20th or 30th file, only to find an error in your prototype—and did you already use this IP address 12 machines ago? You can't recall.

The prototype route loses its sheen on large buildouts because it tries to go two directions at once: it mixes the things that change (hostname, IP address) with those that don't (disk layout, time zone). Kickstart doesn't support macros or replacement variables in *ks.cfg*; but if your target machines load *ks.cfg* via URL, and you have some programming skills, you can write a tool to generate the data on the fly.

To implement such a tool is beyond the scope of this Short Cut, but I can walk through the high-level design. Any such solution would mix a *data store* (the things that change) with a *templating solution* (the things that don't change). The data store would hold the per-machine data, such as the IP address and hostname. You would also need a unique identifier, perhaps the hostname, such that you could pick up a given machine's data. The data store could be a flat file, XML data, or a relational database such as PostgreSQL or MySQL.

Any modern, web-friendly language has a templating solution or 12. Java has Apache Velocity or Freemarker, PHP has Smarty, Perl has Template Toolkit. They all follow the same mail-merge concept: create a template with placeholders for the data, feed that and some data through the templating engine, and get a complete *ks.cfg* in return.

In turn, to invoke the system, you pass a machine's unique identifier as a URL parameter. For example:

```
boot: linux ks=http://your.kickstart.server/gen_config?host=server25
```

In this example, the CGI (or servlet, or whatever) generates a *ks.cfg* for the machine `server25`.

If you don't see the strength of dynamically generated config files, skim David N. Blank-Edelman's *Perl for System Administration*. In Chapter 5, he describes a simple machine inventory system, around which he builds code to generate config files. You could leverage such a system to generate *ks.cfg* on the fly.

Kickstart, running on the target machine, doesn't care what generates the *ks.cfg* data. It can't really tell. As long as it's syntactically correct (to satisfy Kickstart) and logically correct (to build the machine as you need it), the source of the data is irrelevant.

To its credit, Kickstart does support a mild form of dynamic configuration. If you specify a URL that ends in / (as though you're requesting a directory), Kickstart will append a unique identifier that includes the IP address. For example, given the boot string:

```
linux ks=http://your.kickstart.server/configs/
```

and the build-time IP address of `10.10.10.101`, Kickstart will request:

```
http://your.kickstart.server/configs/10.10.10.101-kickstart
```

of the server.

This route has its limitations, notably that you must know the machine's build-time IP address in advance. I personally prefer to use DHCP at build time and specify the permanent IP in *ks.cfg*, but a DHCP service has to keep track of every host's hardware address in order to match the requesting MAC to the provided IP.

If you don't already have the MAC-to-IP accounting in place, ask yourself whether it's worth the effort to implement it just for Kickstart builds. Chances are, you'll save more time and effort manually entering the IP information at build time.

Kickstart Security

Kickstart stores the target machine's root password in *ks.cfg*, which may make a lot of sysadmins uncomfortable. If you serve up *ks.cfg* by URL, someone with a web browser and knowledge of your environment could pick up root access with just a click. There are several ways to address this. Computer security is typically at odds with convenience, however, so be prepared to make a Kickstart build a little less hands-off.

One option is to have a temporary build-time root password that a sysadmin changes immediately after the install. If the sysadmin misses this step of the post-build procedure, that's a potential risk. It's also a hassle for the other sysadmins who expect to access this machine using the standard root password.

You can also limit the scope of people who could possibly find the *ks.cfg* URL by serving up Kickstart files from a private, internal machine instead of a public-facing host. Take this one step further and create a private build network. This doesn't have to be anything fancy. Mix a portable network switch, a couple of cables, and a laptop to serve *ks.cfg* and the install media.

Another option is not to put the root password in *ks.cfg* at all. Kickstart will prompt you for anything it doesn't find in *ks.cfg*, so the build will continue as normal except for that one interruption. This means you're not 100 percent removed from the picture, but depending on your shop's policies, being 99 percent removed may be acceptable.

You can also store *ks.cfg* on removable media, such as a diskette or USB thumb drive. This limits your ability to build several machines simultaneously, though, because you have to connect the thumb drive to every machine as you build it. This is a reasonable compromise if you don't build a lot of machines on a regular basis.

There is no right or wrong answer here. It's all a matter of your shop's policies and how you feel about the tradeoffs between convenience and security.

Kickstart Upgrades

Just as Kickstart installs are easy compared to manual installs, Kickstart upgrades are even easier. That's because there's a lot less for the installer to decide, and hence, less for you to prepare. Just watch out for a few land mines.

About Kickstart upgrades

Similar to OS installs, OS upgrades involve a mass-push of RPMs. Unlike installs, however, upgrades involve fewer decisions. You aren't changing disk layout, nor the root password, nor any other of the machine's base configurations. You're not even adding any new software. Instead, Kickstart will determine what software is on the target machine and install newer versions of those products. In turn, an upgrade *ks.cfg* only involves a handful of parameters:

- Kickstart type (`upgrade` instead of `install`)
- Location of install media (such as `url` or `nfs`)
- Installed language support (`langsupport`)
- Boot loader configuration (`bootloader`—that is, `grub`)
- A few settings that matter only for the duration of the install, such as network configuration and keyboard type.

Only the boot loader configuration is host-specific. Chances are that you can use the same *ks.cfg* to upgrade most or even all of your shop.

Preparing for a Kickstart upgrade

Running a Kickstart-managed OS upgrade is similar to an install: prepare your install media and *ks.cfg*, boot the target machine, and let it run.

The earlier section, Automating the Build, contains instructions on how to set up your tree of install media. Be sure to maintain the proper directory structure.

The *ks.cfg* for upgrades is rather short. Example 3 is an annotated sample.

Example 3. A sample ks.cfg for OS upgrades

```
## choice: upgrade or install
upgrade

## where is our install media?
url--url http://your.kickstart.server/media/FC6-i386-upgrade


## upgrade language
lang en_US.UTF-8

##
## runtime (installed) language support:
## - single language: "lang en_US"
## - multiple languages: "lang--default en_US en_UK"
## - all languages: "lang--default en_US"
##

## language codes are available in
##   /usr/share/redhat-config-language/locale-list
langsupport --default en_US

## keyboard type--for the install
keyboard us

## upgrade in text mode, rather than GUI
text

## reboot automatically after the upgrade
reboot

## this network info is used for the upgrade only
network --bootproto dhcp

## where to put the boot loader?
bootloader --location=mbr

## notice, there's no "%packages" section.  Kickstart
## determines which (OS-bundled) RPMs are installed and
## upgrades them accordingly.
##
## That is to say, you can't choose to upgrade just
## PostgreSQL or gcc; you get everything at once.
```

To boot the target machine, use the boot media from the *new* (upgrade) OS tree. I've learned the hard way that the boot media and the install media are closely related. For example, if you boot from an FC4 CD when you're upgrading to FC5, the process will (rightfully) fail.

That's all there is to it. Grab a test system and upgrade. When the target machine reaches the ever-familiar `boot:` prompt, tell it to Kickstart based on your fancy new *ks.cfg*:

```
boot: linux ks=http://your.kickstart.server/upgrade.cfg
```

You can head out for a drink while Kickstart does the work, and when you return your machine will have a new OS. Be sure to review the upgrade logs (`/root/upgrade.log` and `upgrade.log.syslog`) to make sure there were no problems.

Before You Turn It Loose...

I'd be remiss in my duties as an author to describe just the mechanics of a Kickstart upgrade. That's just part of the story. Although the tool does the heavy lifting, it's still up to you to plan the process.

OS upgrades aren't quite the same as OS installs. In the former case, no one is using the machine, so there's no end-user impact if the install fails. The extent of your damage is restarting the install.

By comparison, an upgrade requires you to take a machine out of service, make some changes, and return it to its end-users. People expect the machine to work as it did before. The less technical they are, the less they'll notice the changes under the hood—unless those changes cause a problem. Kickstart gives you the power to perform several upgrades in short order, so if you're not careful you can have a large mess on your hands before you know it.

Kickstart doesn't have to put you in the dog house, though. All you have to do is set up some safety nets. Your boss (or client, or whomever) won't necessarily thank you for a smooth upgrade, but you'll spare yourself the headaches of post-upgrade trauma.

Get a backout plan

Newton's Third Law applies even to systems administration: for every action, there is an equal and opposite reaction. At least, there *should* be an equal and opposite reaction. You should always be able to revert a machine to its original state following a change. Some people call this a *backout plan*. I prefer the term *time travel*. Time travel is not unlike an insurance policy: you don't need it every day; but when you need it, you're glad to have it.

Consider changing a config file. Time travel is simple: make a backup of the file, or put it in source control, before you make your change. If the change doesn't work, put the old file back until you can figure out the problem.

Time travel for an OS upgrade is similar, but on a larger scale: get a solid, full backup. An upgrade changes every RPM in the system at once (minus your third-party RPMs, of course) so there's no going back once the process starts. If a Fedora Core 5 to 6 upgrade fails partway through, chances are slim that the machine will boot into a workable state of Fedora Core 5 and a half. The only realistic way to bring a machine back to life after a failed upgrade is to wipe it clean and restore the old data. (Before you do that, boot from a rescue CD and fetch the system's `/root/upgrade.log` and `upgrade.log.syslog` to help you track down problems.) Breathe easy, go home on time, try again some other day.

Time travel also helps when the upgrade itself (Kickstart) succeeds, but there's post-upgrade shrapnel. It's not uncommon for third-party software to react poorly to a system change. If that software happens to be your day job's VPN client or your company's primary database engine, you'll have to roll back your changes with a wipe-and-restore dance.

Don't let a backup make you overconfident! You also have to make sure the data is *viable*. If your backup software has a way to verify that your data made it to the media, use it. If at all possible, restore the data to an empty disk to make sure you don't get media read errors.

This is one reason I prefer not to do OS upgrades at all. Instead, I set up a new machine and migrate services—database, web server, whatever—there, one at a time. That lets me run the service in parallel, on both the old and the new hardware, while testing for issues. That also makes my rollback a one-step process: point the end-users to the service running on the old machine.

Of course, such luxuries of hardware aren't always possible. If you're stuck having to upgrade a machine in-place, master the art of time travel so you don't have to pull your hair out.

Tier your shop

My firm belief in finding problems is, "better now than later." That's because a problem uncovered now (whatever that happens to be) typically requires less hassle to fix than that same problem uncovered later.

To that end, categorize your shop's machines and upgrade accordingly. A smaller shop may have only "test" and "production." A larger shop may have more buckets:

- **Crash'n'burn** (test) hosts are the sysadmin playground. No one relies on these machines, so if an upgrade renders one a smoldering heap, you still get to go home on time. Try upgrades on these first, just to make sure the process works as expected.
- If your company creates any software in-house, chances are you have several **development** machines where application teams take their code for a test run. Upgrade these next, and give your developers a chance to address any issues before you move forward.
- Your test team and business partners use **user acceptance test** (UAT, or "QA" for Quality Assurance) machines to try software your developers have written, as well as some third-party-vended products. UAT is a dress rehearsal for production, so the business partners should take this opportunity to determine whether the upgrade impacts their day-to-day work.
- Finally, **production** machines are the real deal. They serve content to your customers and crunch data for your accounting team.

Each phase expands the responsibility of testing to include different parties. Everyone should have had a chance to try the new OS before it moves to production. The production upgrade itself should be a very dull but well-scripted play. You (and your business partners!) want that upgrade to be a non-event.

But what could go wrong?

It's fair to ask what could possibly go wrong in an OS upgrade. Red Hat spends a lot of time testing the software before it's packaged into a release. The Fedora crowd even lets the public poke at several betas before an official birth announcement. Upgrades don't typically destroy non-OS files or zap partitions. If you jump on the bandwagon several weeks after an OS release, then, shouldn't someone have shaken out all of the bugs already?

Maybe. Maybe not. Internal Red Hat testing and prerelease Fedora Test cycles should smoke out egregious errors, such as a broken installer, but even an error-free upgrade can leave some land mines. You won't find those until you run through your own test cycle.

First of all, consider the kernel. It separates the hardware from the rest of the OS, so without it your machine is an expensive doorstop. Expect an OS upgrade to install a new, *stock* kernel. (At least, the stock version of a Red Hat-tweaked kernel.) If you've built a custom kernel—for fun, for quirky hardware, for whatever reason—you'll have to rebuild it against the new OS version's kernel source.

This, by the way, is one reason to stick with stock kernels as much as possible. You'll have an easier upgrade path.

Changes in individual software products may also catch you off-guard. OS upgrades tend to move from one major revision of a product to another. Your favorite command-line tool—the one called from all of your shell scripts—can change its options. Some fancy SQL calls that work in PostgreSQL 7 choke in version 8, and so on. You and your end-users will want to run through any applications to make sure the newer versions don't pack any unwanted surprises.

Software changes can just as easily trip up your developers. Any native code that ran under the previous OS merits a full test suite because changes in libraries (such as OpenSSL or libcURL) can cause some nasty surprises. In a best-case scenario, the code will fail to run because the libraries have changed too much between versions. Worst-case, your native-code apps will *load* just fine (because the library function signatures match between versions) but break somewhere in the middle (because the functions' innards now do something completely different with that `void *` argument).

Configuration files can also complicate post-upgrade life. A change in parameter names or default values can cause mail to bounce or, worse yet, file it under */dev/null*. If you test the new software first, you can catch this before it becomes a problem.

Your upgrades don't have to cause headaches. Once you've identified and mitigated the risks, you can sleep easy while Kickstart does all of the mind-numbing work.

Custom Yum Repo

Between the initial OS install and upgrade, you still have some work to do. Software authors release minor enhancements or bug fixes, called *updates*, and they can be tough to manage by hand. This section describes a tool called *yum* that takes the sting out of managing updates.

With Kickstart to install and upgrade your machines, and yum to update them, you could end up with a lot of free time on your hands.

Note that, as of this writing, yum is supported only in Fedora and CentOS. While it's possible to install yum on a Red Hat Enterprise Linux (RHEL) machine, the supported update method is a tool called `up2date`.

What Is Yum?

Generally speaking, installing and updating RPMs can be a hassle. That's because it's up to you to track down dependencies (and conflicts!) between packages as you install, update, and remove software from your system. Yum is a tool that wraps RPM installs and updates.

For example, suppose that you want to install `gtkpod` to manage your handheld MP3 player. `gtkpod` has several dependencies, though, so unless you already have Perl, Python, and certain MP3 utilities installed, the `rpm` command would fail. You would have to fetch those dependent products' RPMs, and *their* dependent RPMs, and so on until you had satisfied the entire dependency tree. Many failed `rpm -i` calls later, you finally have `gtkpod` installed—or you give up halfway through.

Using yum, this same operation is a one-liner:

```
# yum install gtkpod
```

A few minutes later, `gtkpod` is ready to roll. Yum resolves all of `gtkpod`'s dependencies for you, without so much as a whimper. You hardly lift a finger.

Yum also shines in how it handles updates (sometimes called *patches*). In software terms, an *update* is a minor change to a product, such as a bug fix or closing a security hole. Updates to a software product are between two minor revisions, whereas an *upgrade* is a move from one major release version to another.[1]

Software upgrades typically involve major changes and feature enhancements, and therefore arrive less often than updates. A full-blown OS upgrade is a mass-upgrade of software, whereas an OS update addresses a few software products at a time based on what minor changes are available.

Given what it took to install one RPM manually, you can imagine the hassle of updating several products at once. Using yum, this is another one-liner:

```
# yum update
```

Once again, yum will track down updates, as well as their dependencies, and install them.

How does yum work all of this magic? It first finds the RPMs installed on your system. It compares this to the available RPMs in a *repository* ("repo" for short), which is a collection of RPMs hosted on a remote web site. If there are any newer

[1]Exactly what is a major or minor release number depends on the product. Consider a revision scheme *X.Y.Z*, such as *1.4.12*. A new major-revision release may involve a change in the X or Y, compared to the previous release. A minor revsion may involve a difference in the Y or Z. This is a general guideline, though, and the only way to really tell what is a major release revision is to note the product's documentation.

RPMs (updates) in the repo, yum fetches them from the repo and installs them on your system. If yum can't resolve a product's dependencies based on a repo's contents—that is, the repo's collection is incomplete—it simply doesn't install that particular RPM.

The primary public repos are managed by the Fedora and CentOS teams. A default Red Hat install includes configurations for the public repos. There are also third-party repos, such as Fedora Extras and livna.org, which host software that compliments the default OS install. With a little work, you can manage your own internal yum repos, too.

Why Would I Want My Own Yum Repo?

Every time you invoke yum, it contacts an outside, public repo and fetches anywhere from a few kilobytes to a few hundred megabytes of data. This is suitable for a small home setup with only a couple of machines. With more machines, though, there are several disadvantages to using the public repos.

First on the list is bandwidth. For every machine you update, you have to fetch another copy of the *same data* from the outside world. A typical X.org or OpenOffice.org update can easily top 100 MB, so with only 10 machines you've pulled an entire gigabyte of data from a mirror and through your routers. Using an internal repo, you can save bandwidth: download once from the outside world to your internal repo(s), then as many times as you want from the internal repo host to other internal machines.

Related to bandwidth is speed. No Internet connection can match an internal 100 megabit or gigabit LAN.

Also consider how much more control you get when you run your own private repos. Yum fetches the *latest* version of a package available from a repo, not necessarily the *next* one up from your installed version. You can't use the public repos to update your machines in phases (refer to Tier your shop in the previous section) because the version of product Foo that yum installs to your test systems today may not be the same as the one it pulls into production next week. If your machines fetch from your internal repos, though, *you* determine the latest RPMs that yum sees.

Given those benefits, however, using an internal yum repo doesn't necessarily provide more security. RPMs carry an internal GPG signature that yum and other RPM-related tools verify. If someone tampers with an RPM, a test of its GPG signature will fail whether that package comes from the outside world or an internal yum repo. While you're no *more* secure using your own repo, then, nor are you any *less* secure.

How Do I Set Up an Internal Yum Repo?

Considering all of yum's magic, setting up an internal yum repo is surprisingly painless. You copy RPMs from the public repos, generate some metadata based on those RPMs, and tell client machines where to look for updates. You can serve the repos (ergo, the RPMs) to other machines via HTTP, FTP, or NFS. I'll describe an HTTP setup in this section, though it shouldn't take much to apply those steps to a different setup if that's your preference.

Directory structure

A yum repo is nothing without RPMs. Use `wget` or any other download tool to fetch the files from the public repos. For maximum convenience, I recommend that you wrap this in some sort of cron job, or at least a script you can invoke at will.

Space requirements vary. Reserve a lot of disk space if you wish to serve repos for several OS versions. As of this writing, my internal Fedora Core 4 and 5 repos—mirrors of the base OS, updates, and some third-party sites—are roughly 30 G each. The Fedora Core 6 repo is a much smaller 12 G, because that OS was just recently released.

Adopt a hirarchical directory structure to keep this area neat and flexible. A predictable, standardized setup will make it easier to set up your yum client config files.

I prefer to have one directory per OS version, then one subdirectory for each chip architecture (such as `i386`, `x86_64`, and `ppc`), and then one directory per mirror. Refer to Figure 2 for an example.

Such a deep structure may seem overkill at first, but it lets me transparently add mirrors and chip architectures to my repo tree. For example, I don't run x86_64 machines today, but I'll probably buy a fancy new computer with all of my book royalties....

Next, make this location available via your web server software. If you already have one setup for Kickstart, it's probably easiest to slide the yum repos right alongside that.

Load the URL in a browser to make sure the files are visible at the location you expect. For example:

```
http://your.yum.server/repos/FC6/i386/os-updates/
```

Even a handful of yum repos will use a lot of space. Whether you back up this area is up to you. I'd rather keep an extra 60 or 80 gigabytes of static data on my backup system rather than recreate it. On the other hand, there's nothing in a yum repo

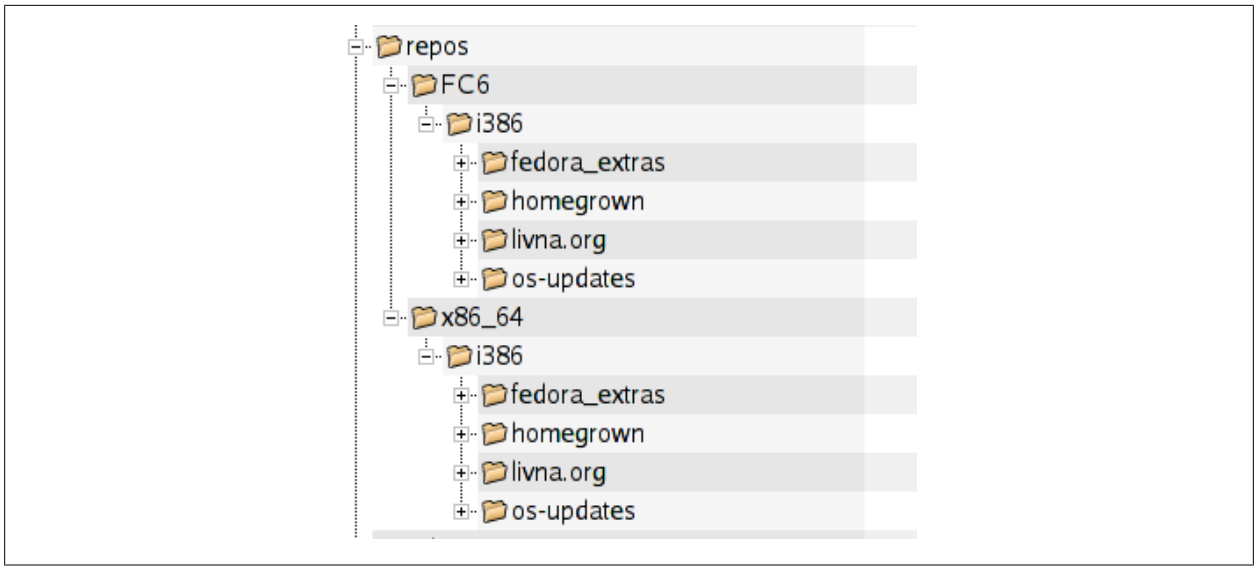

Figure 2. Yum repo directory structure

you can't recreate. You can safely omit the yum repos from your backups if you're prepared to re-download everything in the event of a disk crash (or sysadmin crash, such as an errant `rm -r` or `mkfs`).

Generating metadata

Yum doesn't download all of a repo's RPMs to see what is available. Instead, it fetches special metadata files that explain what a given repo offers.

Use the `createrepo` command to create metadata files for your repo. (Install the `createrepo` RPM on the machine that will host the yum repos, if it isn't already there.) You run `createrepo` on each would-be repo. For example:

```
## create a repo of our OS updates mirror
createrepo /path/to/FC5/i386/os-updates

## ... then one for our Fedora Extras mirror
createrepo /path/to/FC5/i386/fedora_extras

## ... and wrap up with our internal copy of livna.org's offerings
createrepo /path/to/FC5/i386/livna.org
```

`createrepo` will recursively scan the specified directory for RPM files and extract metadata into XML files. It stores those XML files in a subdirectory `repodata` just below the specified directory.

Each `repodata` directory represents a single repo. In the above example, there are three repos for Fedora Core 6, i386 architecture: `os-updates`, `fedora_extras`, and `livna.org`. Have no fear: you don't have to manually point yum to each individual repo. You can configure yum to search several repos at once. Subdividing them as I have demonstrated helps you keep track of a given package's source.

Configuring the clients

The last step is to point your client machines to your shiny new yum repo(s). Note the files in */etc/yum.repos.d*: yum automatically picks up any file with a *.repo* extension, so you don't have to take any extra steps for yum to find your config file.

I suggest that you give your file a descriptive name, such as *internal-defs.repo*, so you and your fellow sysadmins can easily spot it with a glance. I also recommend that you do not change the existing *.repo* files—the ones that shipped with the OS and point to the CentOS or Fedora mirrors—to point to your own repos. This will cause confusion later, because most sysadmins will assume those files have nothing to do with your site-specific repos.

Consider this sample repo config file:

```
[internal-os-updates]
name=Fedora Core $releasever - $basearch - OS updates
baseurl=http://your.yum.server/FC$releasever/$basearch/os-updates
enabled=1
gpgcheck=0


[internal-livna]
name=Fedora Core $releasever - $basearch - livna.org
baseurl=http://your.yum.server/FC$releasever/$basearch/livna.org
enabled=1
gpgcheck=0


[internal-extras]
name=Fedora Core $releasever - $basearch - Fedora Extras
baseurl=http://your.yum.server/FC$releasever/$basearch/fedora_extras
enabled=1
gpgcheck=0
```

This file defines three repositories: one for OS updates (updated RPMs for the base OS), one mirror of the repo at livna.org, and another mirror of Fedora Extras. Use the repository ID, inside square brackets, to enable or disable the repo on the fly using yum's `--enablerepo` or `--disablerepo` flags, respectively. The repo `name` is a human-readable description of this repo.

Yum uses the `baseurl` directive to find the repo. This can be an HTTP (`http://`), FTP (`ftp://`), or filesystem (`file:///`). The `baseurl` points to the base of the repo. That is, yum expects the *repodata* directory to exist just beneath this path. You can specify multiple base URLs, separated by spaces, and yum will try each one. As an alternative, use the `mirrorlist` directive to tell yum to pull the list of URLs from an external file.

Words with the `$` prefix are variables. These are assigned by yum at runtime. `$basearch` expands to the base machine architecture, such as `x86_64` for 64-bit machines or `i386` for machines that run the various Intel-compatible x86 family chips (such as 386, 486, Pentium 586, 686, and so on). `$releasever` is the major release version, such as `5` for Fedora Core 5. Variables help you use a single config file across multiple machines, and multiple versions of the same OS. For example, this sample file will work whether it is run on a Fedora Core 4, 5, or 6 machine. (Refer to the yum documentation for other variables.)

The `enabled` directive tells yum whether to use this repo (value of `1`) or ignore it (`0`). Enable or disable a repo at runtime using yum's `--enablerepo` or `--disablerepo` command-line flags, respectively.

The `gpgcheck` directive tells yum whether to verify an RPM's GPG signature (`1`) or bypass the check (`0`). The value is based on a mix of personal preference and situation. If your RPM download script already checks the signatures as it copies a file to the repo space (that is, it calls `rpm --checksig` on each RPM), then it's reasonable to not bother with the check on the yum client machines. On the other hand, if you are concerned that someone may tamper with your internal repo—which would imply other security problems, by the way—then enable the GPG check. Note that certain third-party RPMs have no signature, so yum will refuse to install those packages if you enable the check.

Next, disable the other repos. (You don't want yum to contact the public mirrors again.) Recall that yum automatically picks up any file with a *.repo* extension in /etc/yum.repos.d. For each repo in those files (except your new, internal repo, of course) specify the `enabled=0` directive. People sometimes define repos inline in /etc/yum.conf, so disable any repos there, too.

Using Your Internal Yum Repo

Before you configure every machine in your shop to point to your new yum repos, take one machine for a test drive. As root on a client machine, run:

```
# yum list
```

...to see what RPMs are available in the repos. Next, enter:

```
# yum update
```

...to have yum fetch and install updated RPMs. This will prompt you before yum updates anything. To bypass the prompt, such as in a script, use yum's `-y` switch:

```
# yum -y update
```

If you have a lot of machines, even typing `yum -y update` can wear you down. yum includes a service and some cron jobs that will happily run the updates for you. To enable them, run the commands as root:

```
# chkconfig --add yum
# chkconfig yum on
# service yum start
```

Fedora Core 6 uses a slightly different method. Instead of a cronjob, it includes the `yum-updatesd` daemon:

```
# chkconfig --add yum-updatesd
# chkconfig yum-updatesd on
# service yum-updatesd start
```

That said, I'm not a fan of letting machines blindly update themselves. Even a minor RPM update merits some level of testing, lest a nasty surprise slip into a production system. The later section Safely Automating Yum explains how to wrap automated yum updates in a layer of change control so you can reap the benefits of both worlds.

Pre-Patching the Kickstart Installation

As much as I like yum, I don't like having to call it immediately after a Kickstart install to patch the OS. Can't I just install the OS with the updates pre-applied? The answer is an emphatic *yes*. This section explains how. (Hint: it's more sophisticated than a call to `yum update` from a postinstall script.)

Eliminate the Middle Man

yum keeps your system up to date between Kickstart's OS installs and upgrades. The Kickstart/yum duo eliminates a lot of your manual labor, but there's still room for improvement. An OS release that's a few months old can have a couple of gigabytes in updates. Building a machine, then, means you have to follow up a Kickstart install with an immediate call to `yum update`.

You could invoke yum from a postinstall script, but that would address just the symptom and not the disease. You would still have to wait for the build-plus-update process to complete before you could release a machine to its end-users. If you're building several machines at once, this scenario also yields a fair amount of network traffic. A more efficient build process would have Kickstart use the updated RPMs in the first place, during the OS install (or upgrade). In other words, Kickstart would use a *prepatched* tree of install media.

This is certainly possible, and with the right tools and a little scripting skill, it's downright easy. You just have to put the updated RPMs in the install tree and

regenerate some metadata. The first step can be tricky, but it's possible to create a tool that does the hard work for you.

Like Kickstart itself, the value of the prepatched solution is its efficiency. If you're building a lot of machines—such as a clone farm, or a nightly rebuild in a lab environment—you would benefit from the savings in time. For target machines that are geographically separate from your Kickstart server—in a remote office, for example—the bandwidth savings will reduce traffic on your long-distance network. You can certainly use a prepatched tree in other, perhaps smaller-scale situations, but you won't reap quite the same ratio of effort to outcome.

Preparing Your Prepatched Install Tree

Directory structure

Directory structure is important if you're building an updated install tree. For one, you must organize your files such that they're easy to find and manage. That means keeping the original OS install files and your yum repos in the same location. Consider the following structure, which builds on the layout I presented in the previous section to include the original OS install files:

```
distros
|
+- FC6
   |
   +- i386
   |  |
   |  +- os
   |  |  |
   |  |  +- Fedora
   |  |     |
   |  |     +- base
   |  |     |
   |  |     +- RPMS
   |  |
   |  +- os-updates
   |  |
   |  +- fedora_extras
   |  |
   |  +- livna.org
   |  |
   |  +- homegrown
   |
   +- x86_64
   |  |
   |  +- ...
   |
   +_ ....
```

The *os* directory holds the original OS install files. In this example, all artifacts related to Fedora Core 6—install media, updates, and third-party yum repos—are in a single place. This isn't a hard requirement for a prepatched install tree, but using such a layout means you won't have to fish around for files when you mix them into a prepatched install tree later.

Second, the directory structure is important in creating the new prepatched tree. This directory must also be web-accessible, and must mirror the layout used by the original install media. The following outlines a directory layout for a prepatched Fedora Core 6 install under i386 architecture. Kickstart clients will point to the *fc6-i386-prepatched-install* directory. Notice that it has the same *Fedora*, *base*, and *RPMS* directories as the original OS install media:

```
+- fc6-i386-prepatched-install
|  |
|  +- Fedora
|     |
|     +- base
|     |
|     +- RPMS
```

(Of course, change the *Fedora* product path to *CentOS* or *RedHat* to match your install media.)

Also notice that I've created a *new* directory for the prepatched install. The original OS files, under *FC6/i386/os* in Figure 1, remain untouched. You want to separate the pristine, original Red Hat content from your own tweaks. The original content serves as your baseline should something go awry.

The *base* directory holds some metadata. Remember that for a moment. First, fill the *RPMS* directory with, well, the RPMS.

Introducing novi: sorting the RPMs

Populating the *RPMS* directory is the tricky part. On one hand, yum gracefully accepts multiple versions of the same RPM in the same tree. If yum sees ProductJMU versions 1.1 and 1.2 in a repo, it will hand you the latter version. By comparison, the Anaconda installer (ergo, Kickstart) doesn't do that kind of math. It wants to see one version of each RPM in the install media tree. If you want a prepatched OS install, you have to help Anaconda along.

One way to do this is to mix the original RPMS and updates into the *RPMS* directory and manually weed out the older versions. Please spare yourself the trouble. You'll lose all of the time you saved by using Kickstart and yum in the first place. Remember how painful it was to track down dependencies for a single RPM (see the earlier section, Custom Yum Repo)? Imagine doing that for all of them.

I say, leave that grunt work to computers. They have no problem crunching through mindless, repetitive tasks while you're at the pub. I didn't find a tool that would create the install tree the way Anaconda needed it, so I wrote it myself and named it *novi*.

The name novi comes from the Russian word *НОВЫЙ*, which means new. (It's common to use the Latin character *y* as an equivalent of the Cyrillic *uri*, or *ы*, but I wanted something a little different.) Novi scans a tree of RPMs and extracts the newest (latest version) of each product. You can use novi to create a prepatched OS install tree.

(There may be other such tools, but I haven't found one yet.)

Novi's command-line syntax is straightforward: point it to one or many trees of RPMs, and tell it where to put the latest-version RPMs of that set. For example:

```
# novi -a {action} \
  -t {toplevel dir of prepatched tree} \
  {directory #1 of RPMs} \
  {directory #2 of RPMs} \
  {more RPM directories...}
```

Given the directory structure described earlier, that would be:

```
# novi -a hardlink \
  -t ..../fc6-i386-prepatched-install/Fedora/RPMS \
  ..../FC6/os/Fedora/RPMS \
  ..../FC6/updates
```

This tells novi to scan the base OS install and updates directories for RPMS. It will sort out the latest version of each product, and hard-link those RPMS to the *Fedora/RPMS* subdirectory of your prepatched install tree. Hard-linking saves space and I/O time, because it creates pointers to the original files instead of copying them.

Novi is available for download from my web site [http://www.ExMachinaTech.net]. The site also hosts novi documentation and examples.

Grab a quick drink while novi churns your hard drive, then generate your metadata.

Similar to yum, Anaconda (ergo, Kickstart) uses files of RPM metadata to determine what software products are in the install tree. The final step in creating a prepatched install tree is to regenerate this metadata based on your new, novi-populated *RPMS* directory.

To start, copy the contents of the *base* directory from the original install tree to your new tree:

```
# cp -ip /path/to/FC6/os/Fedora/base/* \
  /path/to/fc6-i386-prepatched-install/Fedora/base
```

Fedora Core 6 has moved a file outside of the *base* directory. Be sure to copy *images/stage2.img* to your tree, as well:

```
# mkdir /path/to/fc6-i386-prepatched-install/images

# cp -ip /path/to/FC6/os/Fedora/images/stage2.img \
  /path/to/fc6-i386-prepatched-install/images
```

For Fedora Core 5 and 6, run `createrepo` on the top-level directory of your prepatched install tree:

```
# cd /path/to/fc6-i386-install
# createrepo -g Fedora/base/comps.xml ${PWD}
```

The `-g` flag is new. It tells `createrepo` to include some information about the comps file in the usual repo metadata. (If you've been skipping around, see the earlier section Automating the Build for more about this file.) Should an install complain that it can't find "group data," it's really telling you that you forgot to run `createrepo` with the `-g` switch.

The CentOS 4 installer is a little older, more like the Fedora Core 3 setup, so it doesn't use the repo metadata for this step. Instead, it uses the old-style *hdlist* files. To regenerate these, run the `genhdlist` command on the top-level directory of your prepatched install tree:

```
# genhdlist /path/to/centos4-i386-install
```

Note that the hdlist files include RPM filenames in addition to the product names and versions. If your install tells you that it can't find package Foo v1.1.1 and you have Foo v1.1.2 in your prepatched tree, you forgot this step.

Take it for a spin

After all that hard work, why not take your new, prepatched install for a test drive? Grab a test machine and point it to the new tree. Assuming the FC6 examples above, specify the repo URL in your *ks.cfg* file:

```
url --url=http://your.kickstart.server/fc6-i386-install/
```

at the ever-familiar `boot:` prompt. A short while later, your machine will (re)enter the world complete with the latest updates already applied. While the install runs, you can take a few minutes to calculate how much time you've saved by adopting Kickstart, yum, and a prepatched install process.

Safely Automating Yum

Scheduled, unattended updates may yield surprises. (While great for parties and gifts, surprises are often unwelcome in the infrastructure realm.) This section explains how to bask in the convenience of automated yum updates yet mitigate the risks.

Automating Yum's Updates

It's hardly a challenge to run `yum update` now and then on your machines. The more machines you have, though, the more time you spend on accounting: *did I update machine X or machine Y last week? Did I ever update machine Z?*

Yum includes some cron jobs to automatically invoke OS updates. Enable them, and you no longer have to remember to run the updates yourself. Then again, do you really want that? Previous sections have touched on the potential risks of OS updates, and the need for testing. Chances are that you already have a cronjob that pulls updated RPMs from the public repos and into your local yum repos. If you automate the yum updates as well, you lose the testing phase because your machines will just pick out the newest RPMs from the repo. Will tomorrow's (cron'd) `yum update` bring a new quirk or expose an incompatibility with a vended product?

Strange though it may seem, you can have the best of both worlds. By building on some concepts from the previous section, your involvement in enterprise change control boils down to flipping some symbolic links. You just have to treat a collection of updates as a single release.

Cutting a Release

Having spent my career in or around software development teams, I tend to think in terms of product release cycles: "What's still in a test phase?" versus "What's stable, so we can release it to a customer?"

Just about any software, whether shrink-wrapped or downloaded, free or commercial, has a version number. In turn, that version number is tied to a given set of features and maybe some bugs. (The latter is especially true if said product is more than a few weeks old.) A *major* revision ("version 1") has new features and sweeping changes, whereas a *minor* revision ("version 1.1" or even "1.2.13") is one or more bug fixes for a major revision.

The same version concept holds true for operating systems. The catch is that, in the Red Hat world, the OS only gets a major release number: 4, 5, 6, and so on. There's no such thing as Fedora Core 4.2 or 5.1. Instead, the bug fixes within a given release come through the comprised products—the RPMs—not the OS itself. This lack of an explicit service pack level is both a blessing and a curse. The

blessing is that Red Hat doesn't make you wait for a monthly or quarterly all-encompassing update. If Product X has an update that's ready to roll today, Red Hat can release it today.

The curse is that, if you crave change control, you have to roll your own service packs, minor revisions, or whatever you choose to call them. Luckily, this is straightforward and almost easy. Remember the prepatched install tree from the earlier section, Prepatching the Kickstart Installation? Simply tack some identifier, or *label*, on the end of the directory name and voilá you have a release.

There are myriad ways to devise a label. The best guidelines I can give you are to be consistent, and choose a scheme that natually lends itself to easy incrementing —such as letters or numbers.

I personally prefer to use the date as my labels; they represent all of the OS updates up to a certain point in time. For example, if I create an install tree for Fedora Core 5, i386 architecture, with all of the OS updates as of 2006/12/18, I would name that directory *fc5-i386-20061218*.

I would then build a prepatched install tree in this directory, and run `createrepo` at the top to yum-enable it. Then, I could Kickstart new machines and update existing machines to the "Fedora Core 5, i386 architecture, 20061218" release.

As long as I point Kickstart and yum clients to this tree, I can test against just this release. I can download new updates in preparation for a future release, but machines won't see that release tree until I explicitly point them to it.

Adding the Change Control

It's painful (and error-prone) to update the yum client configurations manually every time you cut a new release, though. That's where the magic of symbolic links completes the picture. Create a symbolic link, as from *fc6-i386-latest* to *fc6-i386-20061218* and point all of the yum configs—that is, the `baseurl` directive in the yum repo definition file, in */etc/yum.repos.d*—to the URI `fc5-i386-latest`. The link's target will change when I cut a new release—perhaps on 2006/12/21—but yum is none the wiser. Behind the scenes, fellow sysadmins can glance at the directories and see that we've updated all machines to the 2006/12/18 release.

In a small shop, having just a *-latest* symlink will suffice. Larger, more conservative shops may want a more formal test cycle. This only requires a couple more symbolic links and changes to the machines' yum configurations:

- **Test machines** point directly to the dated directories. Sysadmins manually update the yum configs to test-drive each new release.

- **Development** hosts point to a symlink named *-dev*, such as *fc5-i386-dev*. The yum `baseurl` configuration directive points to the URL:
  ```
  http://your.yum.server/..../fc5-i386-dev
  ```
- **UAT** machines point to a similar symlink, with a *-uat* suffix.
- **Production** hosts point to *-production* symlink. The paranoid among us may even split production into alternate groups, such as *-production-A* and *-production-B*.

In turn, to promote a given release—a dated directory—point one of the named symlinks to it. After you've kicked around release *20061218* in the lab, point the dev link to it, as in *fc6-i386-dev* to *fc6-i386-20061218*.

You can now run `yum update` on a (properly configured) dev machine all you want. The named symbolic link is a set of blinders for yum: it will only see the yum repo tree behind its named symbolic link. Logically following, the same concept applies to UAT and production.

In turn, then, you can safely enable yum's cron jobs. You will never be surprised by an OS update because you know in advance what's in it. You're free to let yum's cron jobs run unattended and, largely, unwatched because your change control is at the server level, not the client level.

Before You Start…

It's easy to let your change control process lull you into a false sense of confidence. There are still two gotchas to automating yum, though they're easy to tackle if you plan ahead.

The first gotcha is related to time. As a sysadmin, you're still on the hook to define and publish an update schedule. *You* know when the cron jobs are set to run, but what about your business partners? Having a regular, well-known maintenance window follows the Principle of Least Surprise: people know that, from 11 p.m. to 4 a.m., the systems may be out of service.

Also, will yum collide with other cron jobs? Try to schedule yum to run after business-critical functions—whether they be end-of-day runs or even your backups—are complete. In the rare event that an update goes awry even after all of your testing, you will have less distracting pressure to recover a system if no one needs it for several more hours.

Second, vended software is sometimes slow to catch up to changes in the OS. Chances are, you don't have much leeway to prod these products' vendors. If your testing reveals that a given OS update conflicts with a vended package, then you should disable that machine's yum cronjobs.

On a related note, this is one more reason to create test environments for your vended products. Similar to an OS upgrade, the only straightforward and reliable means to undo a yum update is to restore from backup. (Yes, you *could* hand-tinker with the RPM system and force some package downgrades, but that may lead to an even greater mess.)

These two gotchas are easy to address before they become an issue. Resolve these in advance, and you can sleep easy while yum does your work for you.

ks.cfg Syntax

This final section describes the *ks.cfg* parameters I mentioned in the earlier section Automating the Build.

For a more detailed reference, please see the Red Hat Linux 9 Customization Guide on Red Hat's web site [http://www.redhat.com/docs/manuals/linux/RHL-9-Manual/custom-guide/s1-kickstart2-options.html]. This document refers to an ancient Red Hat release, but it is nonetheless still relevant.

Installation Type

The first entry in a *ks.cfg* describes the installation type. It is either `install` for a fresh install, or `upgrade` for an upgrade.

Install Media

The keywords `url`, `harddrive`, `nfs` and `cdrom` tell Kickstart where to find the install media.

`url --url={target URL}`

Use this to fetch install media from a remote web server (`http://`) or FTP server (`ftp://`). The web server option is the most flexible of the methods listed here, as it lets you call a server-side program (CGI, Servlet, PHP, etc.) to generate *ks.cfg* on the fly.

`harddrive --partition={partition} --dir={directory}`

Kickstart will mount the disk partition (such as `/dev/hdd1`) and look for the install media under the specified directory. Note that you don't specify the leading `/dev/`, just the device name itself.

`nfs --server={NFS server} --directory={dir}`

Similar to the `harddrive` parameter, except Kickstart mounts an area from the specified NFS server instead of a local disk partition.

`cdrom`

Pull the data from a CD or DVD.

Note that the target URL or directory specifies the top level of the install media. This URL or directory must have the product path (*RedHat*, *Fedora*, or *CentOS*) as an immediate subdirectory.

If your DHCP server doesn't assign nameservers, use raw IP addresses for the `url` and `nfs` installations.

For `nfs` or `url` installs on multihomed target machines, add the `ksdevice` kernel parameter at the boot prompt:

```
linux ks=.... ksdevice=eth1
```

Kickstart will otherwise pause to ask you which NIC to use.

Languages and Input

`lang` and `mouse` indicate the language and mouse type, respectively, the installer will use. By comparison, `langsupport` and `keyboard` set the runtime (installed) language support and keyboard type.

The install-time language and mouse settings don't matter for Kickstart installations since you don't interact with the installer.

On the other hand, if you're manually stepping through an installation (for debug purposes), then enter useful values here. Assuming standard desktop hardware and an English-speaking sysadmin, these values should suffice:

```
lang en_US.UTF-8
mouse generic3ps/2
```

Refer to */usr/share/redhat-config-language/locale-list* for the list of valid languages.

The runtime (permanent) language and keyboard settings certainly do matter. Specify a single language (`en_US`) or a default plus other languages (`--default en_US en_UK fr_FR`). Specifying only a default (`--default en_US`) installs support for all languages.

```
langsupport --default en_US.UTF-8keyboard us
```

Video

For a workstation build you'll likely use `xconfig` to configure your video card and monitor.

```
xconfig --card "VMWare" --videoram 16384 --hsync 31.5-37.9
        --vsync 50-70 --resolution 800×600 --depth 16
```

(I've split the above line for readability; it should be a single line in *ks.cfg*.)

`xconfig` takes the card's name (listed in */usr/share/hwdata/Cards*) and video RAM in kilobytes. The remaining parameters specify the monitor's horizontal and vertical sync rates, resolution, and color depth in bits.

Use the `skipx` directive to skip this step (say, for headless servers). You can manually configure X later.

Networking

The `network` directive configures the target host's network information. If you booted from a DHCP server for a `url` or `nfs` install, `network` will set the machine's permanent configuration. On the other hand, if you load *ks.cfg* from local media, `network` configures the build-time *and* permanent network settings.

Consider the directive:

```
network --device eth0 --bootproto static --ip 10.10.10.237
   --netmask 255.255.255.0 --gateway 10.10.10.254
   --nameserver 10.10.10.11,10.0.0.23,10.1.0.34
   --hostname fc1-test
```

This configures the interface `eth0` with a static IP address of `10.10.10.237`. Notice that the nameserver selection accepts a comma-separated list of IP addresses.

Configure other interfaces by specifying different devices with `--device`. You needn't supply any network information when `--bootproto` is `dhcp` or `bootp`.

Authentication

Set the root password with the `rootpw` directive. There are two forms:

```
rootpw PlainText
rootpw --iscrypted $1$NaCl$X5jRlREy9DqNTCXjHpO75/
```

The first form uses the supplied value as the root password, which it then stores in */etc/shadow* as an MD5-hashed value. The second form, which uses the `--iscrypted` flag, tells Kickstart you have already hashed the password. That is, copy the supplied value verbatim to */etc/shadow*. Using the latter form saves you from exposing your root password directly in *ks.cfg*, but in theory, a person who has access to the MD5-hashed version could still try to reverse it. This is one reason, as mentioned inthe earlier section, Customizing Your Kickstart Install, to use a special build-time root password that you can later reset: it limits your exposure in the event that *ks.cfg* is intercepted.

There are several ways to hash the password in advance, such as copying an existing entry from */etc/shadow* or using OpenSSL's `passwd` module:

```
$ openssl passwd -1 -salt "NaCl" "don't use this"
```

Without the `--iscrypted` flag, Kickstart will use the specified password as is.

On a related note, `authconfig` determines how to authenticate users. This line sets the target host to use MD5-hashed passwords from the local */etc/passwd* and */etc/shadow* files:

```
authconfig --enableshadow --enablemd5
```

There are other authentication options, such as NIS, LDAP, and Kerberos 5.

Firewall

The `firewall` directive sets up a rudimentary ruleset, which is useful for a machine that will talk to the outside world:

```
firewall --enabled --trust=eth0 --http --ssh
```

This rule explicitly trusts traffic from interface `eth0`. The firewall will permit incoming SSH (port 22/tcp) and HTTP (80/tcp) traffic on all interfaces.

Specify `firewall --disabled` to manually configure the firewall later or to skip it altogether. More often than not, it's easier to set up the firewall yourself after the install and per the machine's role(s), rather than trying to piece it together at build time.

Time Zone

Set the machine's time zone with the `timezone` directive:

```
timezone Europe/Paris
```

Valid time zones are in the `TZ` column of the file */usr/share/zoneinfo/zone.tab*.

Boot Loader

The `bootloader` directive sets the location of the GRUB boot loader. This line places it in the master boot record (MBR):

```
bootloader --location=mbr
```

If you don't want a boot loader, specify `--location=none`. Remove an old boot loader from the MBR with the separate `zerombr` directive.

Disks

Disk setup is the most complex part of a *ks.cfg* because there are so many machine- and environment-dependent choices.

`clearpart` removes disk partitions.

```
clearpart --all --drives=sda --initlabel
```

`clearpart` can remove only Linux partitions (`--linux`) or all existing partitions (`--all`). It removes partitions from *all* drives unless you specify the `--drives` flag. The `--initlabel` flag works for previously unused disks or disks with foreign partition

schemes: it clears out the old partitions and sets up a scheme that Linux can understand.

Omit `clearpart` to preserve existing partition boundaries.

`part` sets up partitions. The sample *ks.cfg* uses a simple two-partition layout and has a separate swap partition:

```
part /boot --fstype ext3 --size=100 --ondisk=sda --asprimary
part / --fstype ext3 --size=1024 --grow --ondisk=sda --asprimary
part swap --size=128 --grow --size=256 --ondisk=sda --asprimary
```

The first parameter specifies the mount point, here */boot*, /, and `swap`. (Linux doesn't really mount swap space, but that's a minor technicality.) Set the file-system type with the `--fstype` flag. The sample uses `ext3`. Other options include `ext2` and `vfat` (as used in Windows). Swap doesn't use a file-system type.

Specify a partition's size in megabytes using the `--size` flag. Specify the partition's physical disk with the optional `--ondisk` flag. Mark your primary partitions with `--asprimary`.

`part`'s `--onpart` and `--noformat` flags preserve existing partitions between Kickstart installs. For example, here's how to mount the preexisting `hda7` as */home*:

```
part /home --fstype ext3 --size 1024 --onpart hda7 --noformat
```

Note that this won't shuffle data to another part of the disk if other partition sizes change. Instead, it tells Kickstart to leave *hda7*'s partition boundaries intact and to skip creating a new filesystem there using `mkfs`.

Here is a simple one-disk LVM setup:

```
part /boot --fstype ext3 --size=75 --asprimary
part pv.00 --size=1 --grow --asprimary

volgroup vgroot pv.00
logvol / --name=root.fs --vgname=vgroot --size=1024
logvol swap --name=swap.vol --vgname=vgroot --size=256
```

The second `part` directive sets up a partition as an LVM physical volume (PV). The `--grow` flag grows this partition to the maximum allowable size, so that you needn't know the disk's size ahead of time. `part` still requires a size, though, so it uses a bogus PV partition size of 1 MB.

`logvol` is LVM's `part` equivalent: it accepts the logical volume's mount point and size, in addition to the volume group to which it belongs. `logvol`'s `--name` flag names the volume.

Note that the generated */root/anaconda-ks.cfg* on the target host comments out disk layout.

Rebooting

The `reboot` directive forces the target host to reboot when the installation completes. Don't forget to remove the installation media, lest the machine reboot right back into the installer.

Troubleshooting

Use the `interactive` directive to work through a troublesome Kickstart configuration. Kickstart will use the values from your *ks.cfg* but let you manually click through screens.

Package Selection

The `%packages` directive specifies which RPMs to install on the target host. You may select packages individually or en masse as groups. To specify a group, prefix the name with the `@` symbol and a space. Precede a name with a minus symbol (`-`) to exclude that package from the group.

```
%packages
@ dialup
kernel
grub
e2fsprogs
```

The *Fedora/base/comps.xml* file, from the install media, defines package groups.

Package selection is another area in which it is easiest to perform a manual installation once, then mine the resultant */root/anaconda-ks.cfg* file for information.

Pre- and Postinstall Scripts

The `%pre` and `%post` directives mark the beginning of pre- and postinstall scripts, respectively. See the earlier section Customizing Your Kickstart Install for details.